像 IDE 一样使用 vim

wizardforcel

Published with GitBook

目錄

所需即所获：像 IDE 一样使用 vim

yangyangwithgnu@yeah.net
2016-03-30 16:39:33

【谢谢】

捐赠：支付宝 **yangyangwithgnu@yeah.net** ，支付宝二维码（左），微信二维码（右）

二手书：书，我提高开发技能的重要手段之一，随着职业生涯的发展，书籍也在不断增多，对我而言，一本书最多读三遍，再往后，几乎没有什么营养吸收，这部分书对我已基本无用，但对其他人可能仍有价值，所以，为合理利用资源，我决定低价出售这些书，希望达到两个目的：0）用售出的钱购买更多新书（没当过雷锋的朋友(๑´ڡ`๑)）；1）你低价购得需要的书（虽然二手）。到 https://github.com/yangyangwithgnu/used_books 看看有无你钟意的。

【公告】

- 帮助：英译版编制。**github.com** 上搜索 **vim ide** 关键字后第一匹配项便是本文，洋人浏览到本文的次数非常多，常常收到要求同步发表英文版的邮件，但是，你知道，这 **80+** 页的中文已经耗费我大量业余时间，所以，如果可能，希望有精力的朋友可以将其翻译为英文，感谢！
- 讨论：任何意见建议移步 http://www.v2ex.com/t/138696

【版本】

- v0.1.5，2016-03-30，新增/优化：0）采用 vundle 作为插件管理器，并统一采用各插件在 github.com 的地址；1）C++11/14 语法高亮插件从 STL-Syntax 换为 vim-cpp-enhanced-highlight，后者支持 STL 以及自行编写函数、容器、模版等的高亮；2）接口文件（MyClass.h）与实现文件（MyClass.cpp）快捷切换的插件，从 a.vim 换为 vim-fswitch；3）增加标签系统和语义系统的介绍；4）其他调整；
- v0.1.4，2016-02-16，优化：0）为规避快捷键前导字符重复导致操作等待的问题，优化光标快速移至行首/行尾的快捷键，优化结对符间快速移动的快捷键；1）增加代码折叠的环境恢复。
- v0.1.3，2015-11-08，新增：0）光标快速移至行首的快捷键 lh 与光标右移键 l 冲突，导致光标左移操作等待，现添加 \ 规避该问题；1）中文输入状态导致命令模式无效，借助插件解决该问题。
- v0.1.2，2015-01-18，新增：0）重写“内容查找”，让匹配项具备上下文提醒能力；1）“快速输入结对符”扩充快速选中结对符内文本的相关知识；2）增加支持分支 undo 的介绍；3）增加持久化保存 undo 历史的介绍；4）全文结构调整，将“内容查找”和“内容替换”移至“4 代码分析”，将“快速输入结对符”更名为“快速编辑结对符”，并移至“8 其他辅助”。
- v0.1.1，2014-12-27，新增/修正：0）重写“代码收藏”章节，停用过时的 visual mark，启用用户体验更优的 vim-signature（@arcticlion，谢谢）；1）新增“基于语义的导航”章节，YCM 新增该项功能；2）调整“5.2 模板补全”章节结构，UltiSnips 不再提供预定义代码模板；3）protodef 插件更新，修复 protodef 生成成员函数实现的返回语句错误的问题；4）给出安装插件 vim-instant-markdown 的详细步骤。
- v0.1.0，2014-10-13，新增。发布初始版本。

【目录】

【正文】

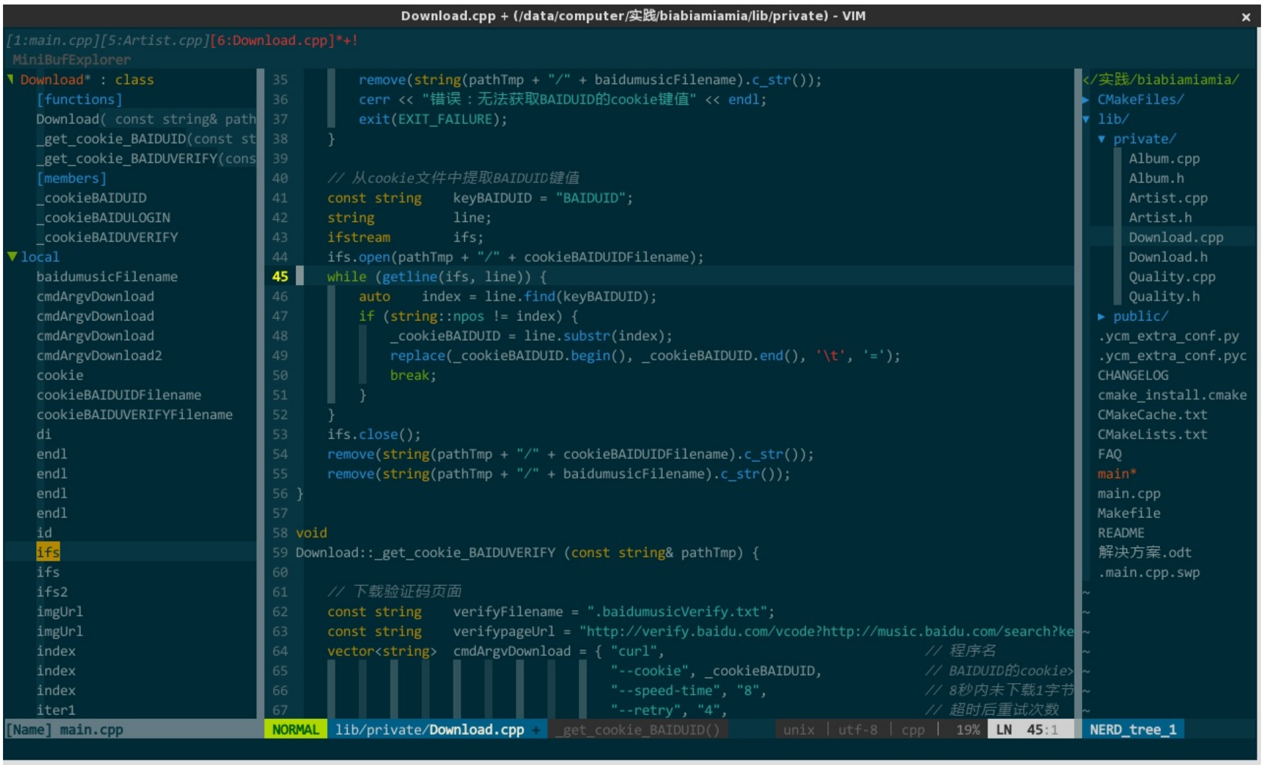
开始前，我假设你：0）具备基本的 vim 操作能力，清楚如何打开/编辑/保存文档、命令与插入模式间切换；1）希望将 vim 打造成 C/C++ 语言的 IDE，而非其他语言。

关于 vim 的优点，你在网上能查到 128+ 项，对我而言，只有两项：0）所思即所得，让手输入的速度跟上大脑思考的速度，1）所需即所获，只有你想不到的功能、没有实现不了的插件。希望获得前者的能力，你需要两本教程深入学习，《Practical Vim: Edit Text at the Speed of Thought》和《vim user manual》；要想拥有后者的能力，通读本文 -。-#。对于 vim 的喜爱，献上湿哥哥以表景仰之情：

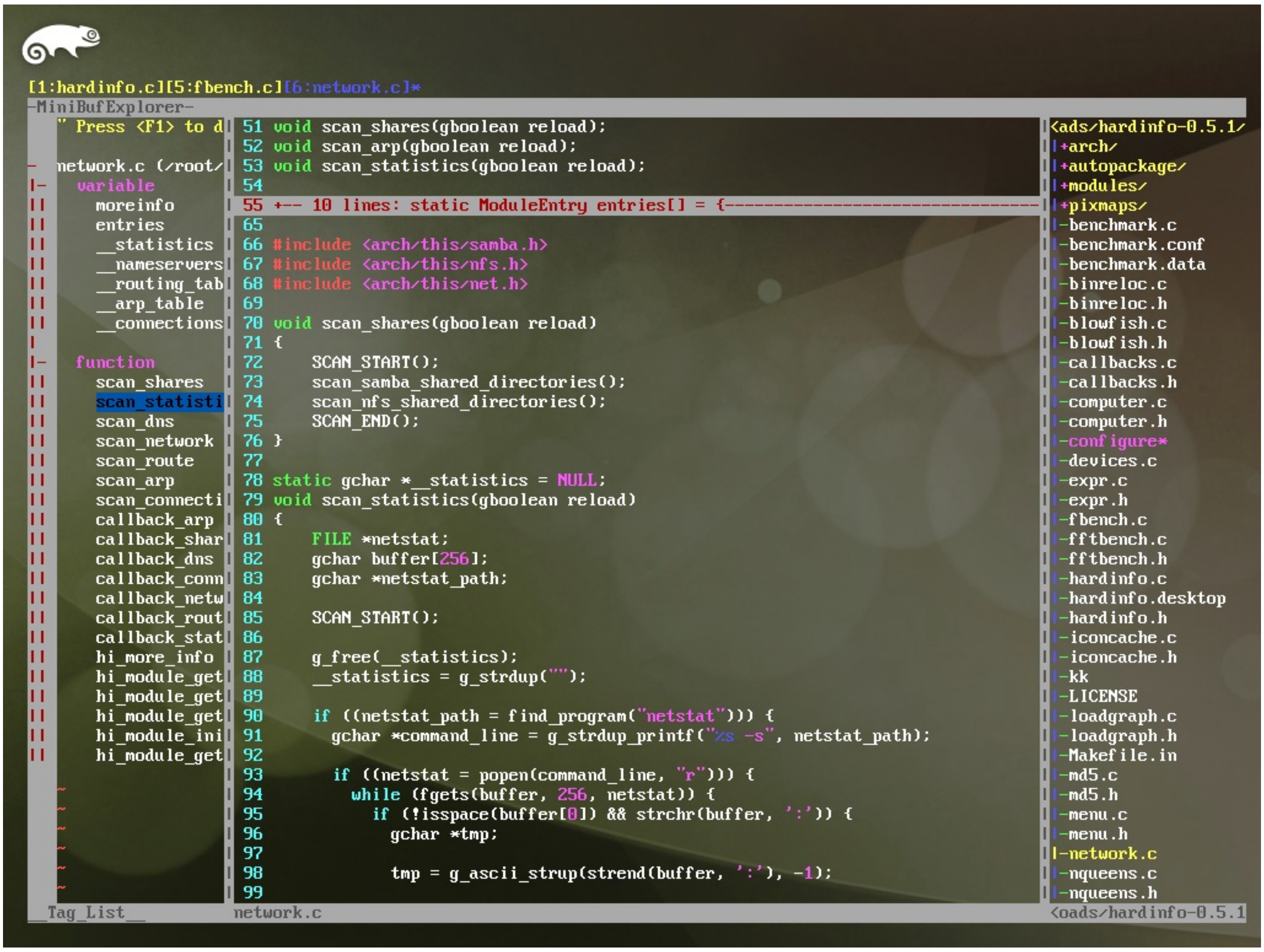
vi 之大道如我心之禅，
vi 之漫路即为禅修，
vi 之命令禅印于心，
未得此道者视之怪诞，
与之为伴者洞其真谛，
长修此道者巨变人生。

作：reddy@lion.austin.com
译：yangyangwithgnu@yeah.net

言归正传，说说 vim 用于代码编写提供了哪些直接和间接功能支撑。vim 用户手册中，50% 的例子都是在讲 vim 如何高效编写代码，由此可见，vim 是一款面向于程序员的编辑器，即使某些功能 vim 无法直接完成，借助其丰富的插件资源，必定可以达成目标，这就是所需即所获。 我是个目标驱动的信奉者，本文内容，我会先给出优秀 C/C++ IDE 应具备哪些功能，再去探索如何通过 vim 的操作或插件来达到目标。最终至少要像这个样子：



（图形环境下 IDE 总揽）



（纯字符模式下 IDE 总揽）

0 vim 必知会

在正式开始前先介绍几个 vim 的必知会，这不是关于如何使用而是如何配置 vim 的要点，这对理解后续相关配置非常有帮助。

0.1 .vimrc 文件

.vimrc 是控制 vim 行为的配置文件，位于 ~/.vimrc，不论 vim 窗口外观、显示字体，还是操作方式、快捷键、插件属性均可通过编辑该配置文件将 vim 调教成最适合你的编辑器。

很多人之所以觉得 vim 难用，是因为 vim 缺少默认设置，甚至安装完后你连配置文件自身都找不到，不进行任何配置的 vim 的确难看、难用。不论用于代码还是普通文本编辑，有必要将如下基本配置加入 .vimrc 中。

前缀键。各类 vim 插件帮助文档中经常出现 \，即，前缀键。vim 自带有很多快捷键，再加上各类插件的快捷键，大量快捷键出现在单层空间中难免引起冲突，为缓解该问题，引入了前缀键 \，这样，键 r 可以配置成 r、\r、\\r 等等多个快捷键。前缀键是 vim 使用率较高的一个键（最高的当属 Esc），选一个最方便输入的键作为前缀键，将有助于提高编辑效率。找个无须眼睛查找、无须移动手指的键 —— 分号键，挺方便的，就在你右手小指处：

```
" 定义快捷键的前缀，即<Leader>
let mapleader=";"
```

既然前缀键是为快捷键服务的，那随便说下快捷键设定原则：不同快捷键尽量不要有同序的相同字符。比如，\e 执行操作 0 和 \eb 执行操作 1，在你键入 \e 后，vim 不会立即执行操作 0，而是继续等待用户键入 b，即便你只想键入 \e，vim 也不得不花时间等待输入以确认是哪个快捷键，显然，这让 \e 响应速度变慢。\ea 和 \eb 就没问题。

文件类型侦测。允许基于不同语言加载不同插件（如，C++ 的语法高亮插件与 python 的不同）：

```
" 开启文件类型侦测
filetype on
" 根据侦测到的不同类型加载对应的插件
filetype plugin on
```

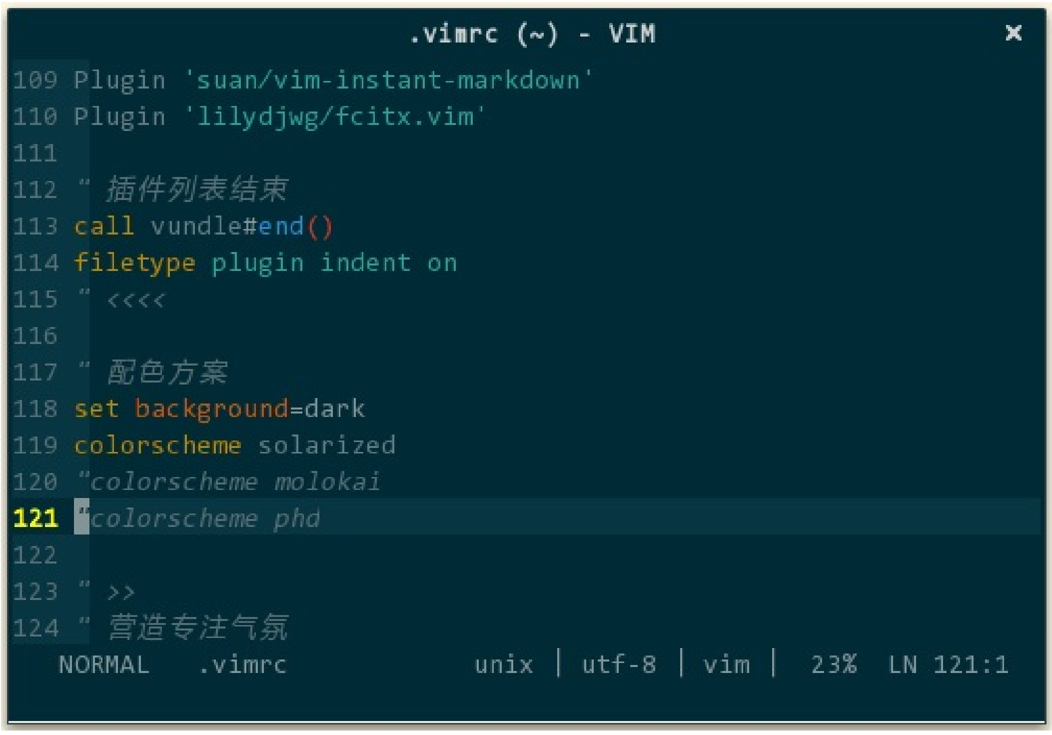
快捷键。把 vim（非插件）常用操作设定成快捷键，提升效率：

```
" 定义快捷键到行首和行尾
nmap LB 0
nmap LE $
" 设置快捷键将选中文本块复制至系统剪贴板
vnoremap <Leader>y "+y
" 设置快捷键将系统剪贴板内容粘贴至 vim
nmap <Leader>p "+p
" 定义快捷键关闭当前分割窗口
nmap <Leader>q :q<CR>
" 定义快捷键保存当前窗口内容
nmap <Leader>w :w<CR>
" 定义快捷键保存所有窗口内容并退出 vim
nmap <Leader>WQ :wa<CR>:q<CR>
" 不做任何保存，直接退出 vim
nmap <Leader>Q :qa!<CR>
" 依次遍历子窗口
nnoremap nw <C-W><C-W>
" 跳转至右方的窗口
nnoremap <Leader>lw <C-W>l
" 跳转至左方的窗口
nnoremap <Leader>hw <C-W>h
" 跳转至上方的子窗口
nnoremap <Leader>kw <C-W>k
" 跳转至下方的子窗口
nnoremap <Leader>jw <C-W>j
" 定义快捷键在结对符之间跳转
nmap <Leader>M %
```

立即生效。全文频繁变更 .vimrc，要让变更内容生效，一般的做法是先保存 .vimrc 再重启 vim，太繁琐了，增加如下设置，可以实现保存 .vimrc 时自动重启加载它：

```
" 让配置变更立即生效
autocmd BufWritePost $MYVIMRC source $MYVIMRC
```

比如，我可以随时切换配色方案：



（配置变更立即生效）

其他。搜索、vim 命令补全等设置：

```
" 开启实时搜索功能
set incsearch
" 搜索时大小写不敏感
set ignorecase
" 关闭兼容模式
set nocompatible
" vim 自身命令行模式智能补全
set wildmenu
```

以上的四类配置不仅影响 vim，而且影响插件是否能正常运行。很多插件不仅要在 .vimrc 中添加各自特有的配置信息，还要增加 vim 自身的配置信息，在后文的各类插件介绍中，我只介绍对应插件特有配置信息，当你发现按文中介绍操作后插件未生效，很可能是 vim 自身配置信息未添加，所以一定要把上述配置拷贝至到你的 .vimrc 中，再对照本文介绍一步步操作。.vimrc 完整配置信息参见附录，每个配置项都有对应注释。另外，由于有些插件还未来得及安装，在你实验前面的插件是否生效时，vim 可能有报错信息提示，先别理会，安装完所有插件后自然对了。

0.2 .vim/ 目录

.vim/ 目录是存放所有插件的地方。vim 有一套自己的脚本语言 vimscript，通过这种脚本语言可以实现与 vim 交互，达到功能扩展的目的。一组 vimscript 就是一个 vim 插件，vim 的很多功能都由各式插件实现。此外，vim 还支持 perl、python、lua、ruby 等主流脚本语言编写的插件，前提是 vim 源码编译时增加 ---enable-perlinterp、--enable-pythoninterp、--enable-luainterp、--enable-rubyinterp 等选项。vim.org 和 github.com 有丰富的插件资源，任何你想得到的功能，如果 vim 无法直接支持，那一般都有对应的插件为你服务，有需求时可以去逛逛。

vim 插件目前分为 *.vim 和 *.vba 两类，前者是传统格式的插件，实际上就是一个文本文件，通常 someplugin.vim（插件脚本）与 someplugin.txt（插件帮助文件）并存在一个打包文件中，解包后将 someplugin.vim 拷贝到 ~/.vim/plugin/ 目录，someplugin.txt 拷贝到 ~/.vim/doc/ 目录即可完成安装，重启 vim 后刚安装的插件就已经生效，但帮助文件需执行 :helptags ~/.vim/doc/ 才能生效，可通过 :h someplugin 查看插件帮助信息。传统格式插件需要解包和两次拷贝才能完成安装，相对较繁琐，所以后来又出现了 *.vba 格式插件，安装便捷，只需在 shell 中依次执行如下命令即可：

```
vim someplugin.vba
:so %
:q
```

不论是直接拷贝插件到目录，还是通过 *.vba 安装，都不便于插件卸载、升级，后来又出现了管理插件的插件 vundle。

后面就正式开始了喽，文中前后内容顺序敏感，请依次查阅。

1 源码安装编辑器 vim

发行套件的软件源中预编译的 vim 要么不是最新版本，要么功能有阉割，有必要升级成全功能的最新版，当然，源码安装必须滴：

```
git clone git@github.com:vim/vim.git
cd vim/
./configure --with-features=huge --enable-pythoninterp --enable-rubyinterp --enable-luain
make
make install
```

其中，--enable-pythoninterp、--enable-rubyinterp、--enable-perlinterp、--enable-luainterp 等分别表示支持 ruby、python、perl、lua 编写的插件，--enable-gui=gtk2 表示生成采用 GNOME2 风格的 gvim，--enable-cscope 支持 cscope，--with-python-config-dir=/usr/lib/python2.7/config/ 指定 python 路径（先自行安装 python 的头文件 python-devel），这几个特性非常重要，影响后面各类插件的使用。注意，你得预先安装相关依赖库的头文件，python-devel、python3-devel、ruby-devel、lua-devel、libX11-devel、gtk-devel、gtk2-devel、gtk3-devel、ncurses-devel，如果缺失，源码构建过程虽不会报错，但最终生成的 vim 很可能缺失某些功能。构建完成后在 vim 中执行

```
:echo has('python')
```

若输出 1 则表示构建出的 vim 已支持 python，反之，0 则不支持。

2 插件管理

既然本文主旨在于讲解如何通过插件将 vim 打造成中意的 C/C++ IDE，那么高效管理插件是首要解决的问题。

vim 自身希望通过在 .vim/ 目录中预定义子目录管理所有插件（比如，子目录 doc/ 存放插件帮助文档、plugin/ 存放通用插件脚本），vim 的各插件打包文档中通常也包含上述两个（甚至更多）子目录，用户将插件打包文档中的对应子目录拷贝至 .vim/ 目录即可完成插件的安装。一般情况下这种方式没问题，但我等重度插件用户，.vim/ 将变得混乱不堪，至少存在如下几个问题：

- 插件名字冲突。所有插件的帮助文档都在 doc/ 子目录、插件脚本都在 plugin/ 子目录，同个名字空间下必然引发名字冲突；
- 插件卸载易误。你需要先知道 doc/ 和 plugin/ 子目录下哪些文件是属于该插件的，再逐一删除，容易多删/漏删。

我希望每个插件在 .vim/ 下都有各自独立子目录，这样需要升级、卸载插件时，直接找到对应插件目录变更即可；另外，我希望所有插件清单能在某个配置文件中集中罗列，通过某种机制实现批量自动安装/更新/升级所有插件。vundle（https://github.com/VundleVim/Vundle.vim）为此而生，它让管理插件变得更清晰、智能。

vundle 会接管 .vim/ 下的所有原生目录，所以先清空该目录，再通过如下命令安装 vundle：

```
git clone https://github.com/VundleVim/Vundle.vim.git ~/.vim/bundle/Vundle.vim
```

接下来在 .vimrc 增加相关配置信息：

```
" vundle 环境设置
filetype off
set rtp+=~/.vim/bundle/Vundle.vim
" vundle 管理的插件列表必须位于 vundle#begin() 和 vundle#end() 之间
call vundle#begin()
Plugin 'VundleVim/Vundle.vim'
Plugin 'altercation/vim-colors-solarized'
Plugin 'tomasr/molokai'
Plugin 'vim-scripts/phd'
Plugin 'Lokaltog/vim-powerline'
Plugin 'octol/vim-cpp-enhanced-highlight'
Plugin 'nathanaelkane/vim-indent-guides'
Plugin 'derekwyatt/vim-fswitch'
Plugin 'kshenoy/vim-signature'
Plugin 'vim-scripts/BOOKMARKS—Mark-and-Highlight-Full-Lines'
Plugin 'majutsushi/tagbar'
Plugin 'vim-scripts/indexer.tar.gz'
Plugin 'vim-scripts/DfrankUtil'
Plugin 'vim-scripts/vimprj'
Plugin 'dyng/ctrlsf.vim'
Plugin 'terryma/vim-multiple-cursors'
Plugin 'scrooloose/nerdcommenter'
Plugin 'vim-scripts/DrawIt'
Plugin 'SirVer/ultisnips'
Plugin 'Valloric/YouCompleteMe'
Plugin 'derekwyatt/vim-protodef'
Plugin 'scrooloose/nerdtree'
Plugin 'fholgado/minibufexpl.vim'
Plugin 'gcmt/wildfire.vim'
Plugin 'sjl/gundo.vim'
Plugin 'Lokaltog/vim-easymotion'
Plugin 'suan/vim-instant-markdown'
Plugin 'lilydjwg/fcitx.vim'
" 插件列表结束
call vundle#end()
filetype plugin indent on
```

其中，每项

```
Plugin 'dyng/ctrlsf.vim'
```

对应一个插件（这与 go 语言管理不同代码库的机制类似），后续若有新增插件，只需追加至该列表中即可。vundle 支持源码托管在 https://github.com/ 的插件，同时 vim 官网 http://www.vim.org/ 上的所有插件均在 https://github.com/vim-scripts/ 有镜像，所以，基本上主流插件都可以纳入 vundle 管理。具体而言，仍以 ctrlsf.vim 为例，它在 .vimrc 中配置信息为 dyng/ctrlsf.vim，vundle 很容易构造出其真实下载地址 https://github.com/dyng/ctrlsf.vim.git ，然后借助 git 工具进行下载及安装。

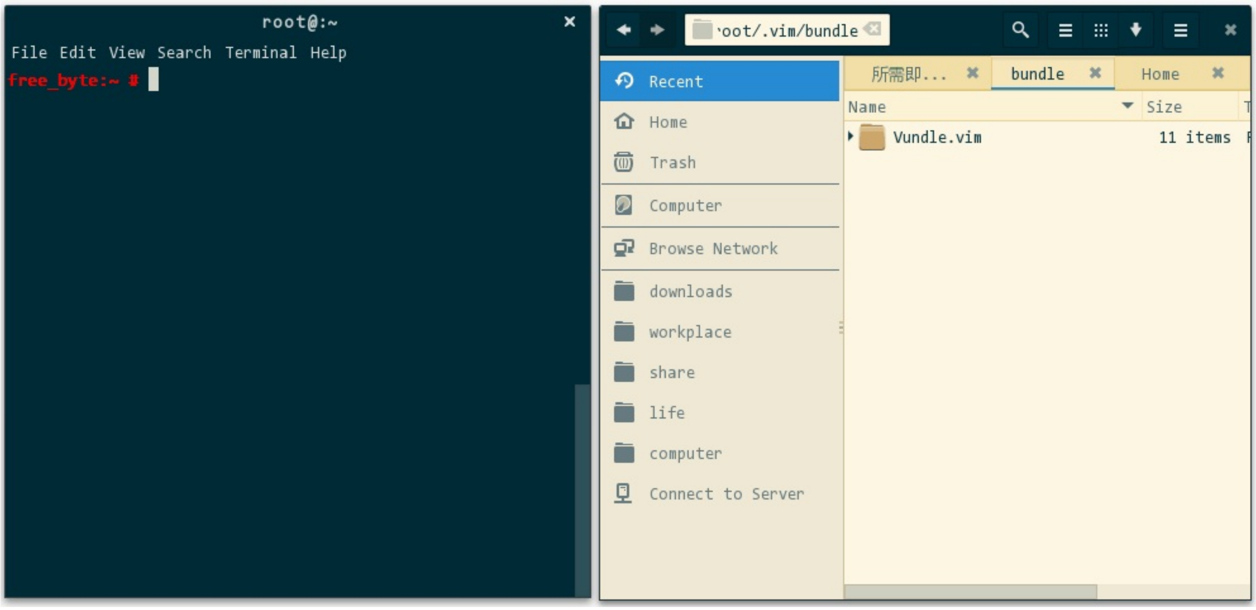
此后，需要安装插件，先找到其在 github.com 的地址，再将配置信息其加入 .vimrc 中的call vundle#begin() 和 call vundle#end() 之间，最后进入 vim 执行

```
:PluginInstall
```

便可通知 vundle 自动安装该插件及其帮助文档。比如，我在 .vimrc 中添加了 4 个插件：

```
Plugin 'VundleVim/Vundle.vim'
Plugin 'altercation/vim-colors-solarized'
Plugin 'Lokaltog/vim-powerline'
Plugin 'octol/vim-cpp-enhanced-highlight'
```

当前，.vim/ 为空，当我在 vim 中执行 :PluginInstall 时，整个自动安装过程便开始了：



（vundle 批量安装插件）

要卸载插件，先在 .vimrc 中注释或者删除对应插件配置信息，然后在 vim 中执行

```
:PluginClean
```

即可删除对应插件。插件更新频率较高，差不多每隔一个月你应该看看哪些插件有推出新版本，批量更新，只需执行

```
:PluginUpdate
```

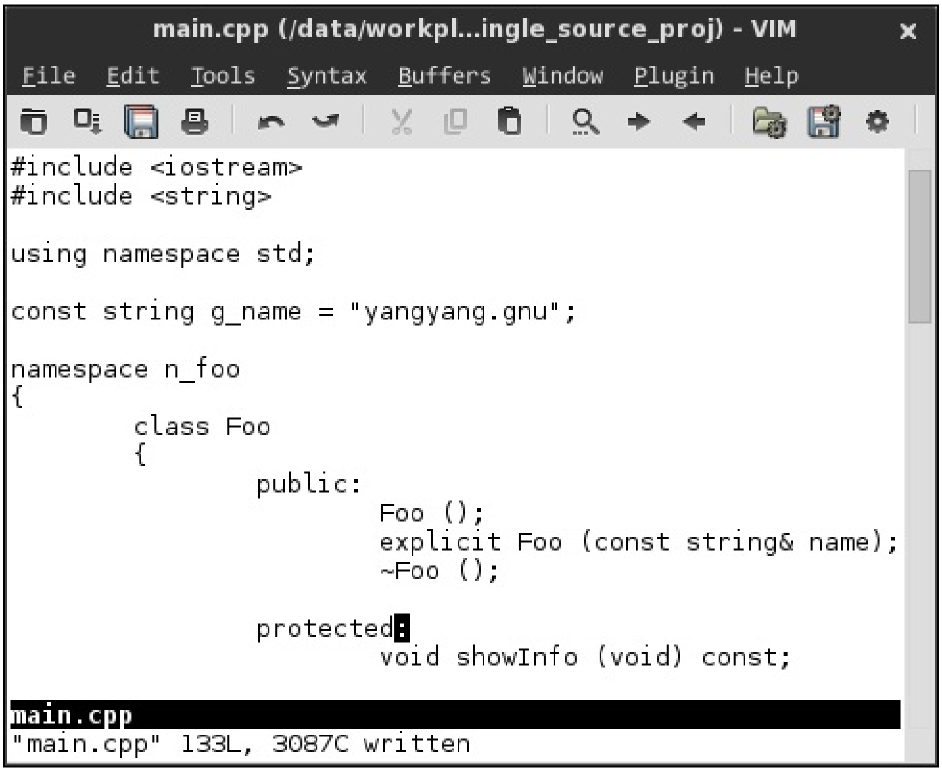
即可。

你得注意插件的下载源。同名插件在 github.com 上可能有多个，比如，indexer 插件，至少就有 https://github.com/vim-scripts/indexer.tar.gz 、https://github.com/everzet/vim-indexer 、https://github.com/shemerey/vim-indexer 等三个，到底应该选哪个呢？以我的经验来看，对于钟意的插件，我会先找其作者的个人网站，上面通常会罗列出托管在 github.com 的具体地址；若没有，我会找该插件在 vim.org 的页面，上面也会有 github.com 托管地址；若还是没有，再以 github 和插件名作为关键字搜索，点赞数多的，通常是你想找的。为节约你找插件地址的时间，本文中出现的每个插件我都会附上其地址。非特殊情况，后文介绍到的插件不再累述如何安装。

通过 vundle 管理插件后，切勿通过发行套件自带的软件管理工具安装任何插件，不然 .vim/ 又要混乱了。

3 界面美化

玉不琢不成器，vim 不配不算美。刚安装好的 vim 朴素得吓人，这是与我同时代的软件么？

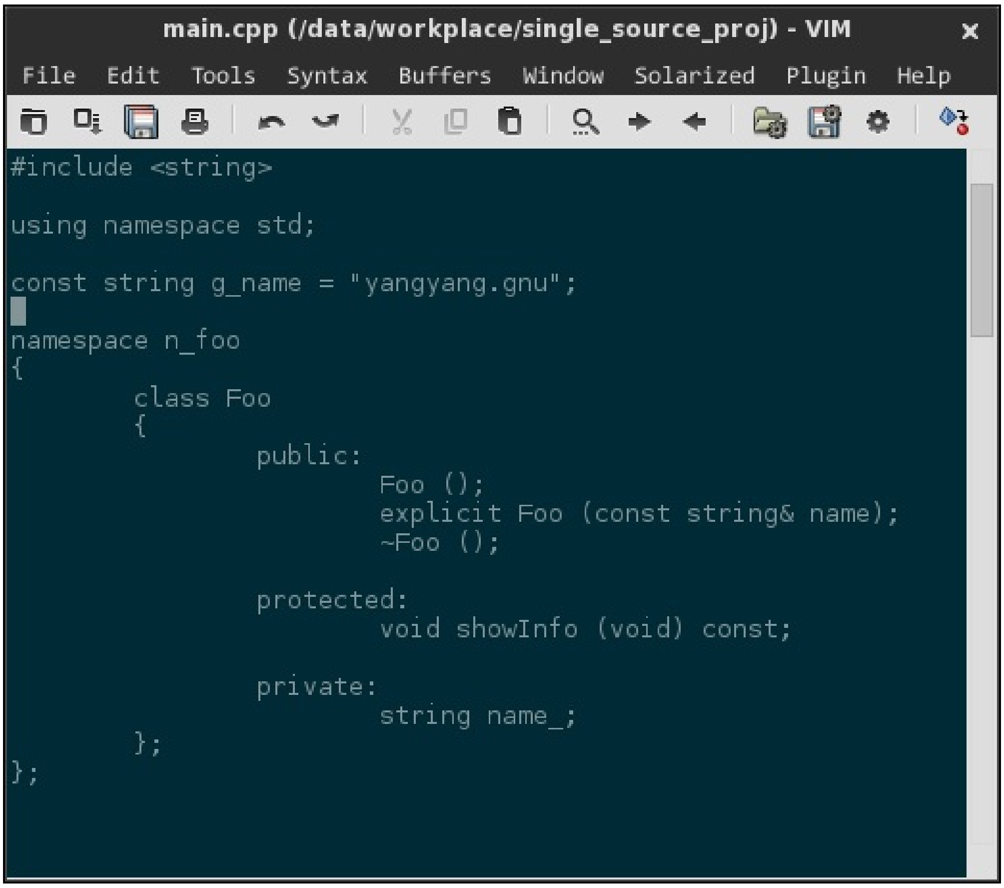


（默认 vim 界面）

就我的审美观而言，至少有几个问题：语法高亮太单薄、主题风格太简陋、窗口元素太冗余、辅助信息太欠缺。

3.1 主题风格

一套好的配色方案绝对会影响你的编码效率，vim 内置了 10 多种配色方案供你选择，GUI 下，可以通过菜单（Edit -> Color Scheme）试用不同方案，字符模式下，需要你手工调整配置信息，再重启 vim 查看效果（csExplorer 插件，可在字符模式下不用重启即可查看效果）。不满意，可以去 http://vimcolorschemetest.googlecode.com/svn/html/index-c.html 慢慢选。我自认为“阅美无数”，目前最夯三甲： * 素雅 solarized（https://github.com/altercation/vim-colors-solarized ） * 多彩 molokai（https://github.com/tomasr/molokai ） * 复古 phd（http://www.vim.org/scripts/script.php?script_id=3139 ） 在 .vimrc 中选用某个主题： ``` " 配色方案 set background=dark colorscheme solarized "colorscheme molokai "colorscheme phd ``` 其中，不同主题都有暗/亮色系之分，这样三种主题六种风格，久不久换一换，给你不一样的心情：



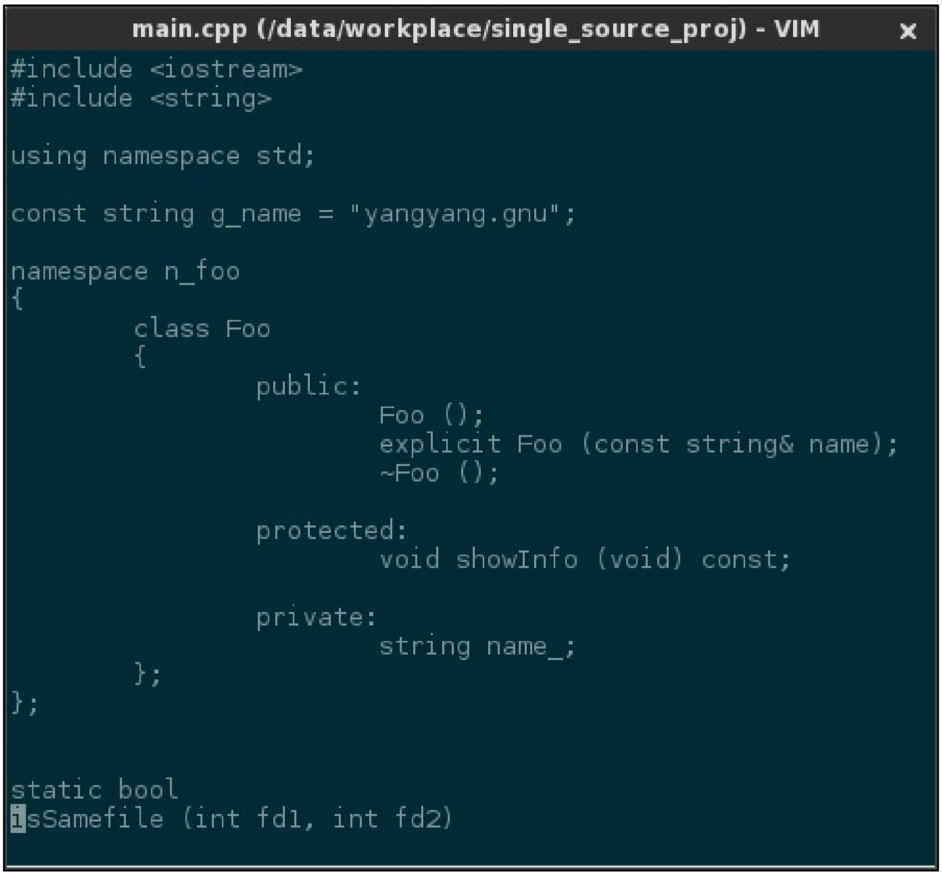
（solarized 主题风格）

3.2 营造专注氛围

如今的 UX 设计讲究的是内容至上，从 GNOME3 的变化就能看出。编辑器界面展示的应全是代码，不应该有工具条、菜单、滚动条浪费空间的元素，另外，编程是种精神高度集中的脑力劳动，不应出现闪烁光标、花哨鼠标这些分散注意力的东东。配置如下：

```
" 禁止光标闪烁
set gcr=a:block-blinkon0
" 禁止显示滚动条
set guioptions-=l
set guioptions-=L
set guioptions-=r
set guioptions-=R
" 禁止显示菜单和工具条
set guioptions-=m
set guioptions-=T
```

重启 vim 后效果如下：



（去除冗余窗口元素）

还容易分神？好吧，我们把 vim 弄成全屏模式。vim 自身无法实现全屏，必须借助第三方工具 wmctrl，一个控制窗口 XYZ 坐标、窗口尺寸的命令行工具。先自行安装 wmctrl，再在 .vimrc 中增加如下信息：

```
" 将外部命令 wmctrl 控制窗口最大化的命令行参数封装成一个 vim 的函数
fun! ToggleFullscreen()
    call system("wmctrl -ir " . v:windowid . " -b toggle,fullscreen")
endf
" 全屏开/关快捷键
map <silent> <F11> :call ToggleFullscreen()<CR>
" 启动 vim 时自动全屏
autocmd VimEnter * call ToggleFullscreen()
```

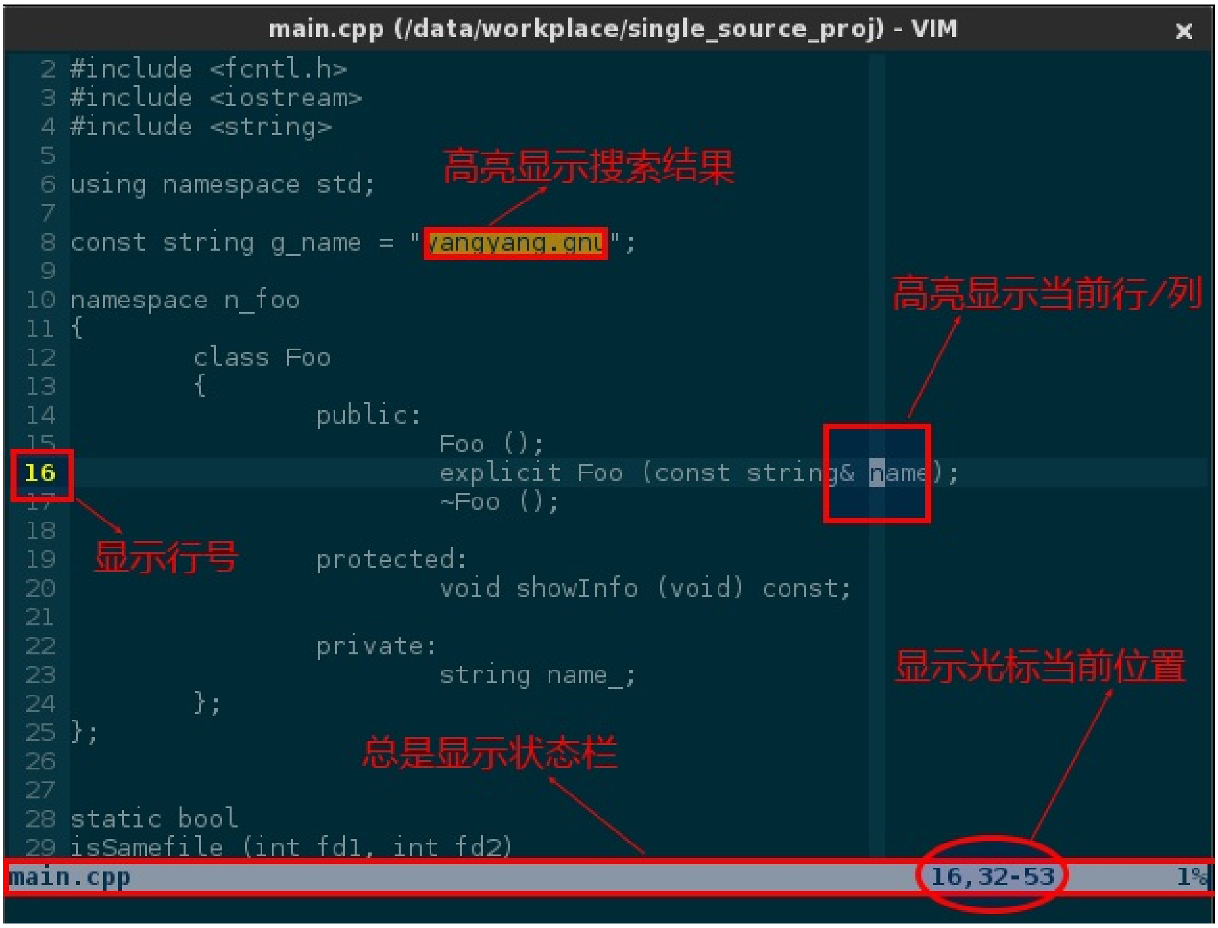
上面是一段简单的 vimscript 脚本，外部命令 wmctrl 及其命令行参数控制将指定窗口 windowid（即，vim）全屏，绑定快捷键 F11 实现全屏/窗口模式切换（linux 下各 GUI 软件约定使用 F11 全屏，最好遵守约定），最后配置启动时自动全屏。

3.3 添加辅助信息

去除了冗余元素让 vim 界面清爽多了，为那些实用辅助信息腾出了空间。光标当前位置、显示行号、高亮当前行/列等等都很有用：

```
" 总是显示状态栏
set laststatus=2
" 显示光标当前位置
set ruler
" 开启行号显示
set number
" 高亮显示当前行/列
set cursorline
set cursorcolumn
" 高亮显示搜索结果
set hlsearch
```

效果如下：



（添加辅助信息）

3.4 其他美化

默认字体不好看，挑个自己喜欢的，前提是你得先安装好该字体。中文字体，我喜欢饱满方正的（微软雅黑），英文字体喜欢圆润的（Consolas），vim 无法同时使用两种字体，怎么办？有人制作发布了一款中文字体用微软雅黑、英文字体用 Consolas 的混合字体 —— yahei consolas hybrid 字体，号称最适合中国程序员使用的字体，效果非常不错（本文全文采用该字体）。在 .vimrc 中设置下：

```
" 设置 gvim 显示字体
set guifont=YaHei\ Consolas\ Hybrid\ 11.5
```

其中，由于字体名存在空格，需要用转义符“\”进行转义；最后的 11.5 用于指定字体大小。

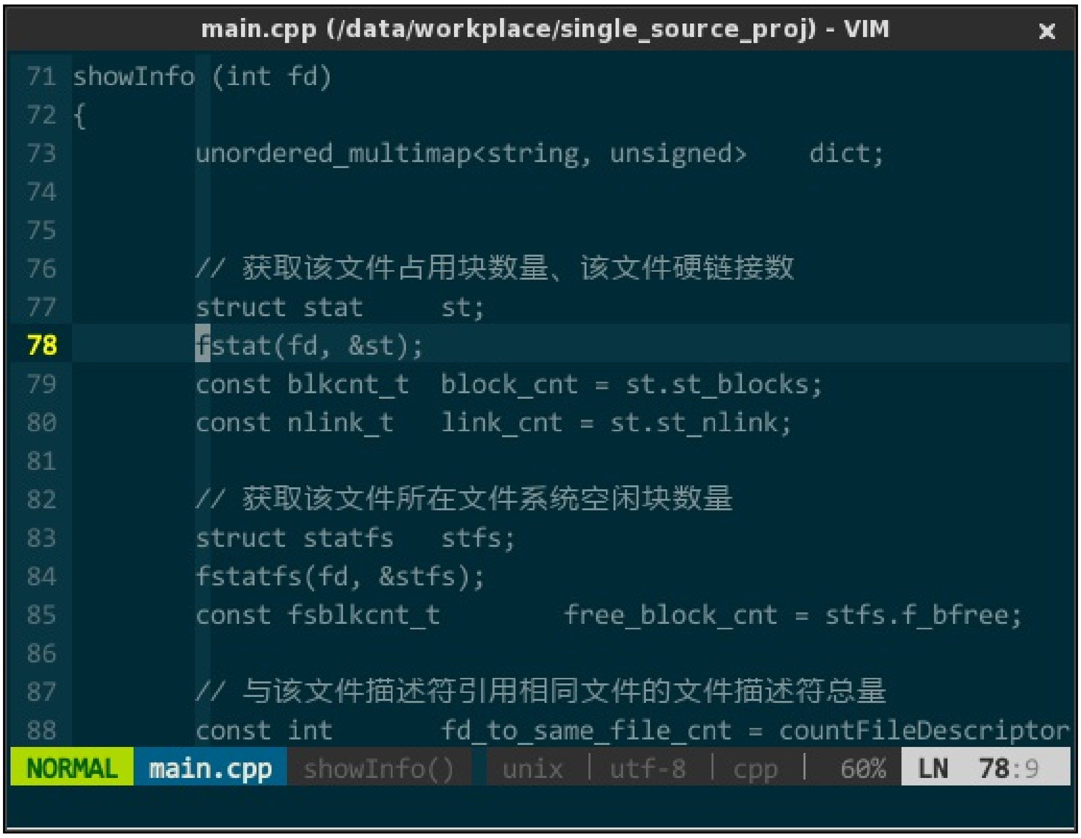
代码折行也不太美观，禁止掉：

```
" 禁止折行
set nowrap
```

前面介绍的主题风格对状态栏不起作用，需要借助插件 Powerline（https://github.com/Lokaltog/vim-powerline ）美化状态栏，在 .vimrc 中设定状态栏主题风格：

```
" 设置状态栏主题风格
let g:Powerline_colorscheme='solarized256'
```

效果如下：



（界面美化最终效果）

图中，中英文混合字体看着是不是很舒服哈；增强后的状态栏，不仅界面漂亮多了，而且多了好些辅助信息（所在函数名、文件编码格式、文件类型）。

4 代码分析

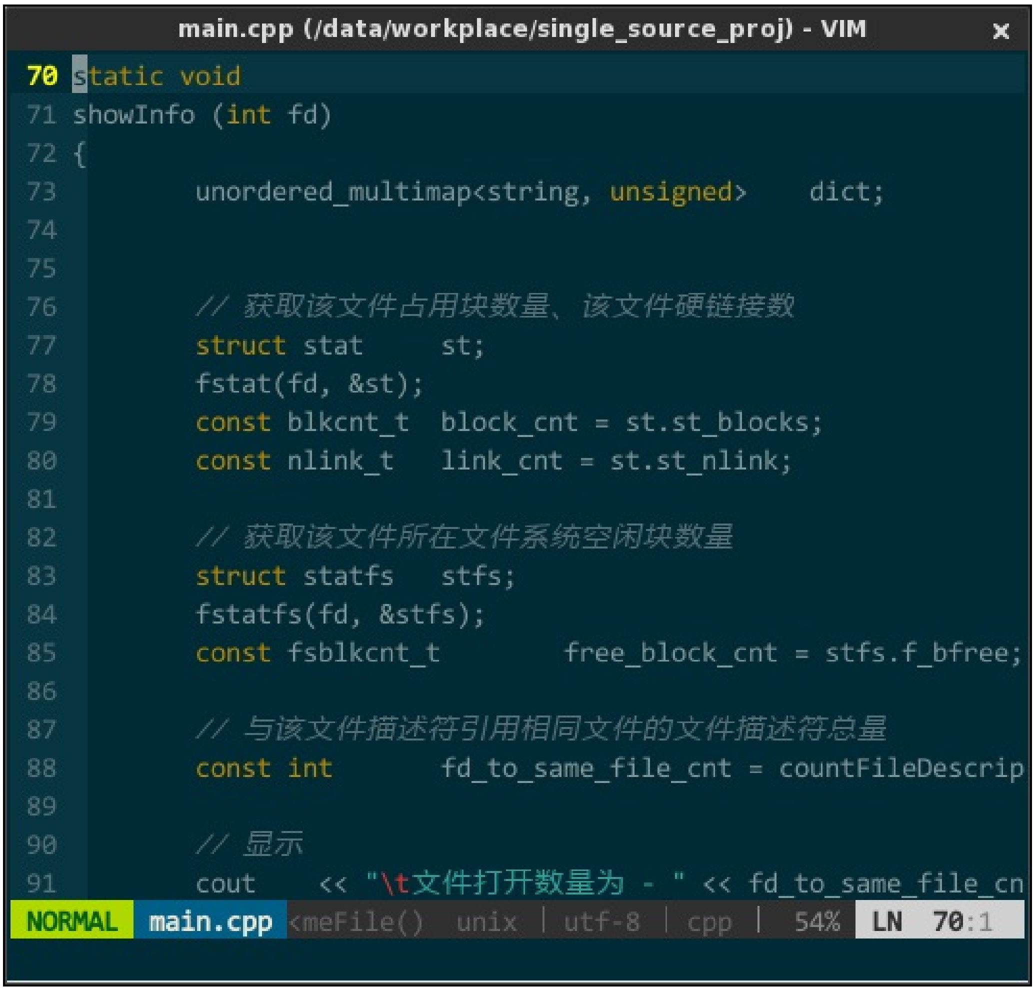
阅读优秀开源项目源码是提高能力的重要手段，营造舒适、便利的阅读环境至关重要。

4.1 语法高亮

代码只有一种颜色的编辑器，就好像红绿灯只有一种颜色的路口，全然无指引。现在已是千禧年后的十年了，早已告别上世纪六、七十年代黑底白字的时代，即使在字符模式下编程（感谢伟大的 fbterm），我也需要语法高亮。所幸 vim 自身支持语法高亮，只需显式打开即可：

```
" 开启语法高亮功能
syntax enable
" 允许用指定语法高亮配色方案替换默认方案
syntax on
```

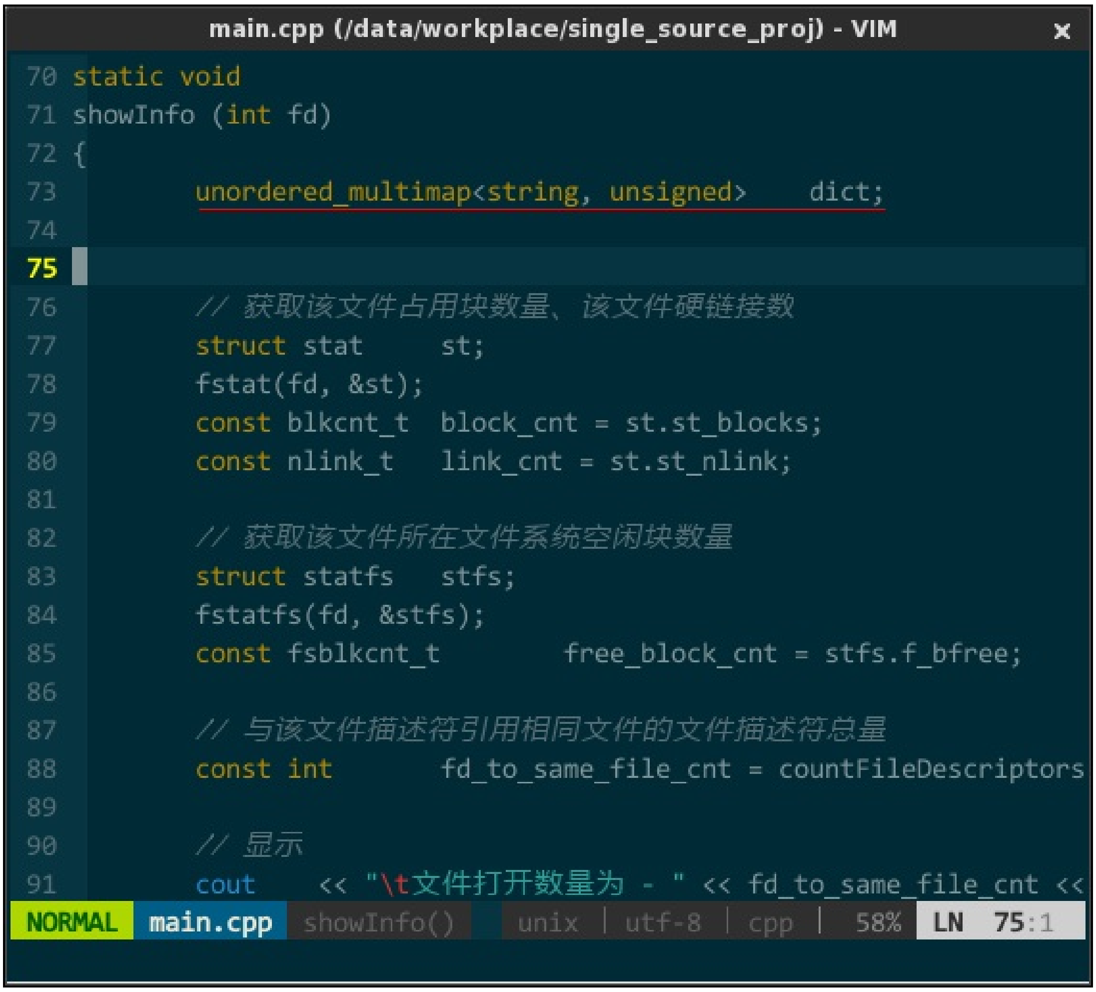
效果如下：



（语法高亮）

上图中 STL 容器模板类 unordered_multimap 并未高亮，对滴，vim 对 C++ 语法高亮支持不

够好（特别是 C++11/14 新增元素），必须借由插件 vim-cpp-enhanced-highlight（https://github.com/octol/vim-cpp-enhanced-highlight ）进行增强。效果如下：



（增强 C++11 及 STL 的语法高亮）

vim-cpp-enhanced-highlight 主要通过 .vim/bundle/vim-cpp-enhanced-highlight/after/syntax/cpp.vim 控制高亮关键字及规则，所以，当你发现某个 STL 容器类型未高亮，那么将该类型追加进 cpp.vim 即可。如，initializer_list 默认并不会高亮，需要添加 ``` syntax keyword cppSTLtype initializer_list ```

4.2 代码缩进

C/C++ 中的代码执行流由复合语句控制，如 if(){} 判断复合语句、for(){} 循环符号语句等等，这势必出现大量缩进。缩进虽然不影响语法正确性，但对提升代码清晰度有不可替代的功效。 在 vim 中有两类缩进表示法，一类是用 1 个制表符（'\t'），一类是用多个空格（' '）。两者并无本质区别，只是源码文件存储的字符不同而已，但，缩进可视化插件对两类缩进显示方式不同，前者只能显示为粗块，后者可显示为细条，就我的审美观而言，选后者。增加如下配置信息： ``` " 自适应不同语言的智能缩进 filetype indent on " 将制表符扩展为空格 set expandtab " 设置编辑时制表符占用空格数 set tabstop=4 " 设置格式化时制表符占用空格数 set shiftwidth=4 " 让 vim 把连续数量的空格视为一个制表符 set softtabstop=4 ``` 其中，注意下 expandtab、tabstop 与 shiftwidth、softtabstop、retab： * expandtab，把制表符转换为多个空格，具体空格数量参考 tabstop 和 shiftwidth 变量； * tabstop 与 shiftwidth 是有区别的。

tabstop 指定我们在插入模式下输入一个制表符占据的空格数量，linux 内核编码规范建议是 8，看个人需要；shiftwidth 指定在进行缩进格式化源码时制表符占据的空格数。所谓缩进格式化，指的是通过 vim 命令由 vim 自动对源码进行缩进处理，比如其他人的代码不满足你的缩进要求，你就可以对其进行缩进格式化。缩进格式化，需要先选中指定行，要么键入 = 让 vim 对该行进行智能缩进格式化，要么按需键入多次 < 或 > 手工缩进格式化； * softtabstop，如何处理连续多个空格。因为 expandtab 已经把制表符转换为空格，当你要删除制表符时你得连续删除多个空格，该设置就是告诉 vim 把连续数量的空格视为一个制表符，即，只删一个字符即可。通常应将这tabstop、shiftwidth、softtabstop 三个变量设置为相同值； 另外，你总会阅读其他人的代码吧，他们对制表符定义规则与你不同，这时你可以手工执行 vim 的 retab 命令，让 vim 按上述规则重新处理制表符与空格关系。 很多编码规范建议缩进（代码嵌套类似）最多不能超过 4 层，但难免有更多层的情况，缩进一多，我那个晕啊：

```
static bool
isSamefile (int fd1, int fd2)
{
    struct stat stat1, stat2;

    if (-1 == fstat(fd1, &stat1) || -1 == fstat(fd2, &stat2)) {
        for () {
            int i;

            while () {
                int j;

                if () {
                    int k;

                    do {
                        int l;
                    } while ();
                }
            }
        }
    }

    return (stat1.st_dev == stat2.st_dev && stat1.st_ino == stat2.st_ino);
}
```

（多层缩进）

我希望有种可视化的方式能将相同缩进的代码关联起来，Indent Guides（https://github.com/nathanaelkane/vim-indent-guides ）来了。安装好该插件后，增加如下配置信息：

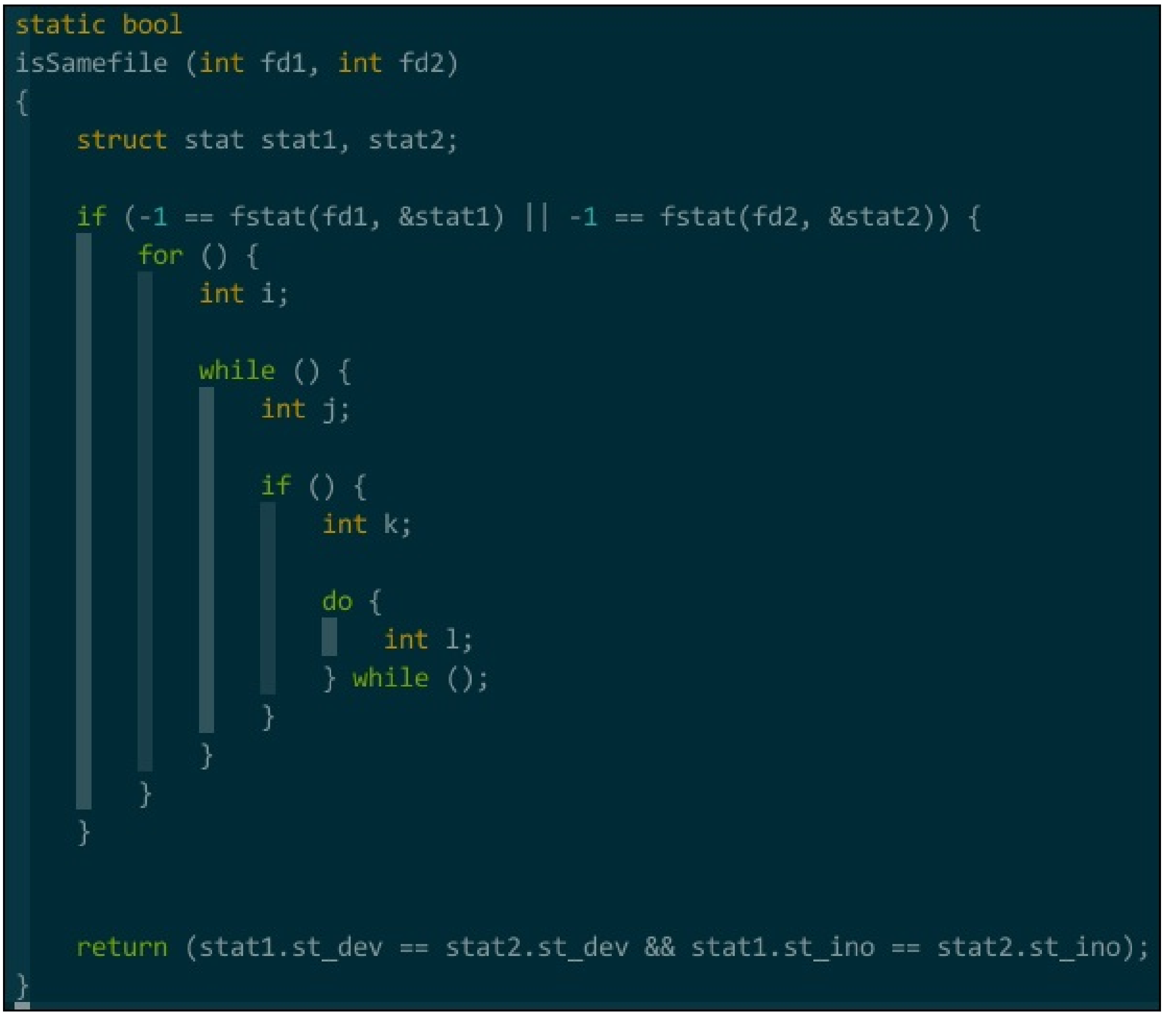
```
" 随 vim 自启动
let g:indent_guides_enable_on_vim_startup=1
" 从第二层开始可视化显示缩进
let g:indent_guides_start_level=2
" 色块宽度
let g:indent_guides_guide_size=1
" 快捷键 i 开/关缩进可视化
:nmap <silent> <Leader>i <Plug>IndentGuidesToggle
```

重启 vim 效果如下：



（不连续的缩进可视化）

断节？Indent Guides 通过识别制表符来绘制缩进连接线，断节处是空行，没有制表符，自然绘制不出来，算是个小 bug，但瑕不掩瑜，有个小技巧可以解决，换行-空格-退格：



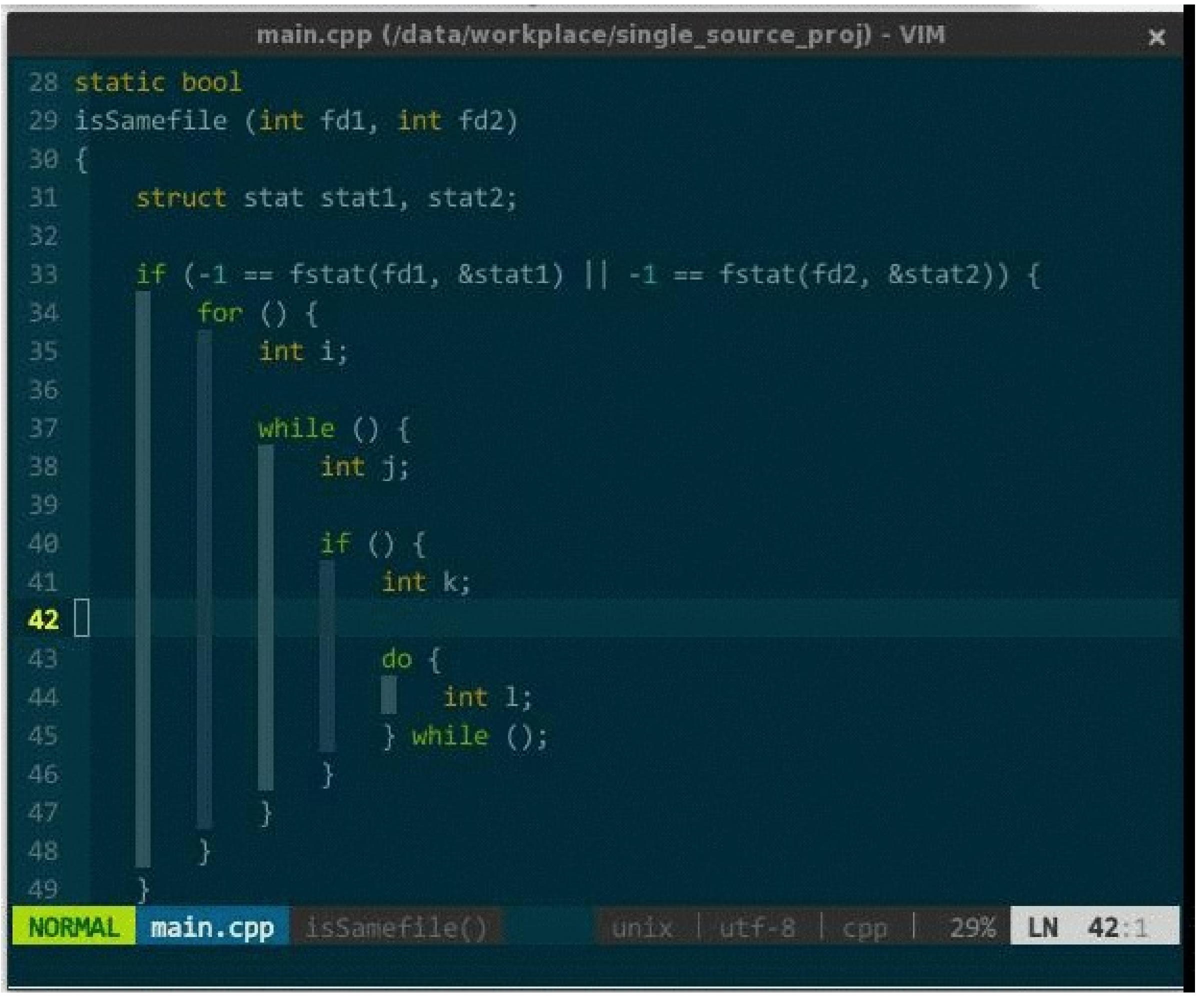
（完美可视化缩进）

4.3 代码折叠

有时为了去除干扰，集中精力在某部分代码片段上，我会把不关注部分代码折叠起来。vim 自身支持多种折叠：手动建立折叠（manual）、基于缩进进行折叠（indent）、基于语法进行折叠（syntax）、未更改文本构成折叠（diff）等等，其中，indent、syntax 比较适合编程，按需选用。增加如下配置信息：

```
" 基于缩进或语法进行代码折叠
"set foldmethod=indent
set foldmethod=syntax
" 启动 vim 时关闭折叠代码
set nofoldenable
```

操作：za，打开或关闭当前折叠；zM，关闭所有折叠；zR，打开所有折叠。效果如下：



（代码折叠）

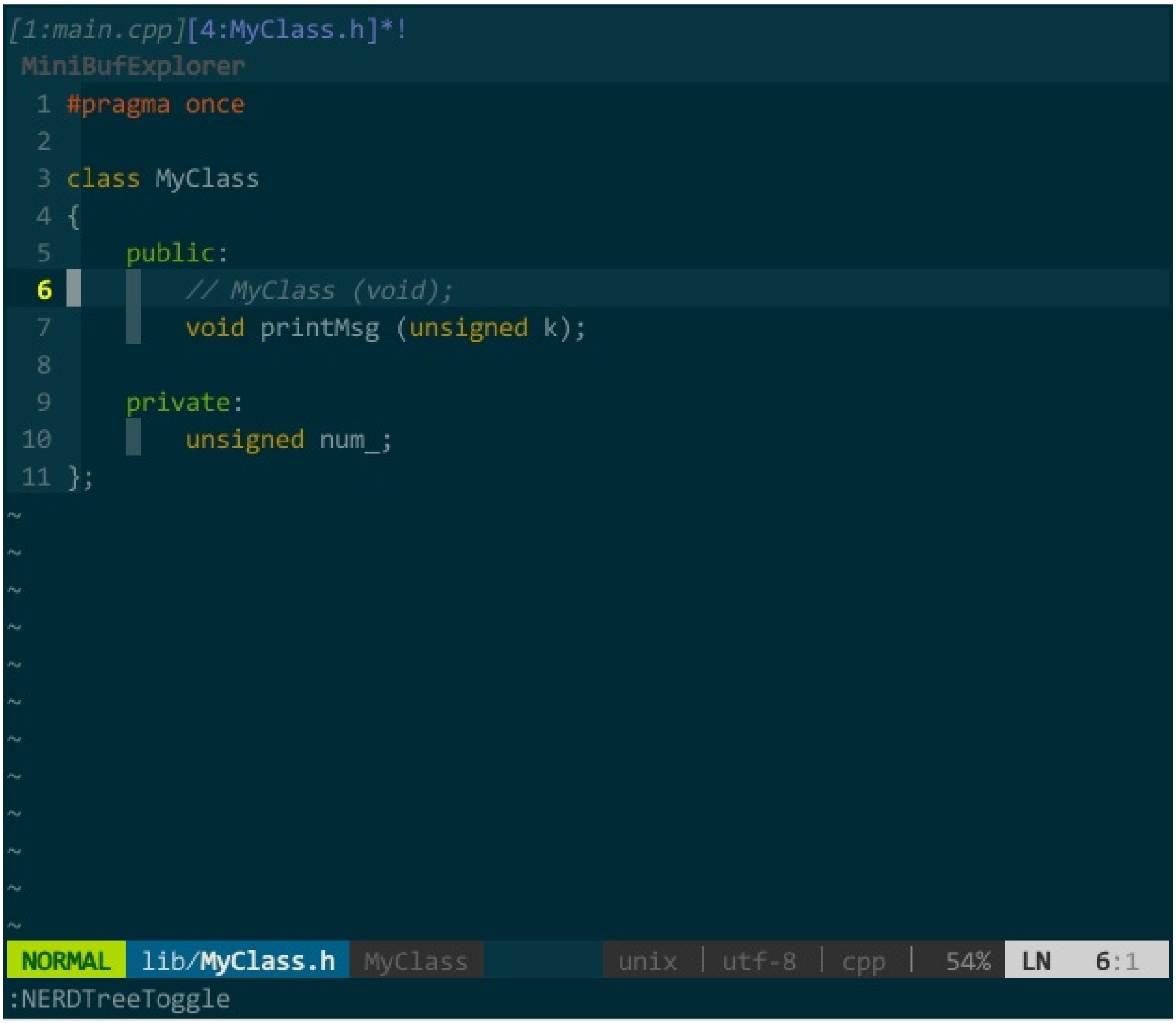
4.4 接口与实现快速切换

我习惯把类的接口和实现分在不同文件中，常有在接口文件（MyClass.h）和实现文件（MyClass.cpp）中来回切换的操作。你当然可以先分别打开接口文件和实现文件，再手动切换，但效率不高。我希望，假如在接口文件中，vim 自动帮我找到对应的实现文件，当键入快捷键，在新 buffer 中打开对应实现文件。

vim-fswitch（https://github.com/derekwyatt/vim-fswitch ）来了。安装后增加配置信息：

```
" *.cpp 和 *.h 间切换
nmap <silent> <Leader>sw :FSHere<cr>
```

这样，键入 ;sw 就能在实现文件和接口文件间切换。如下图所示：



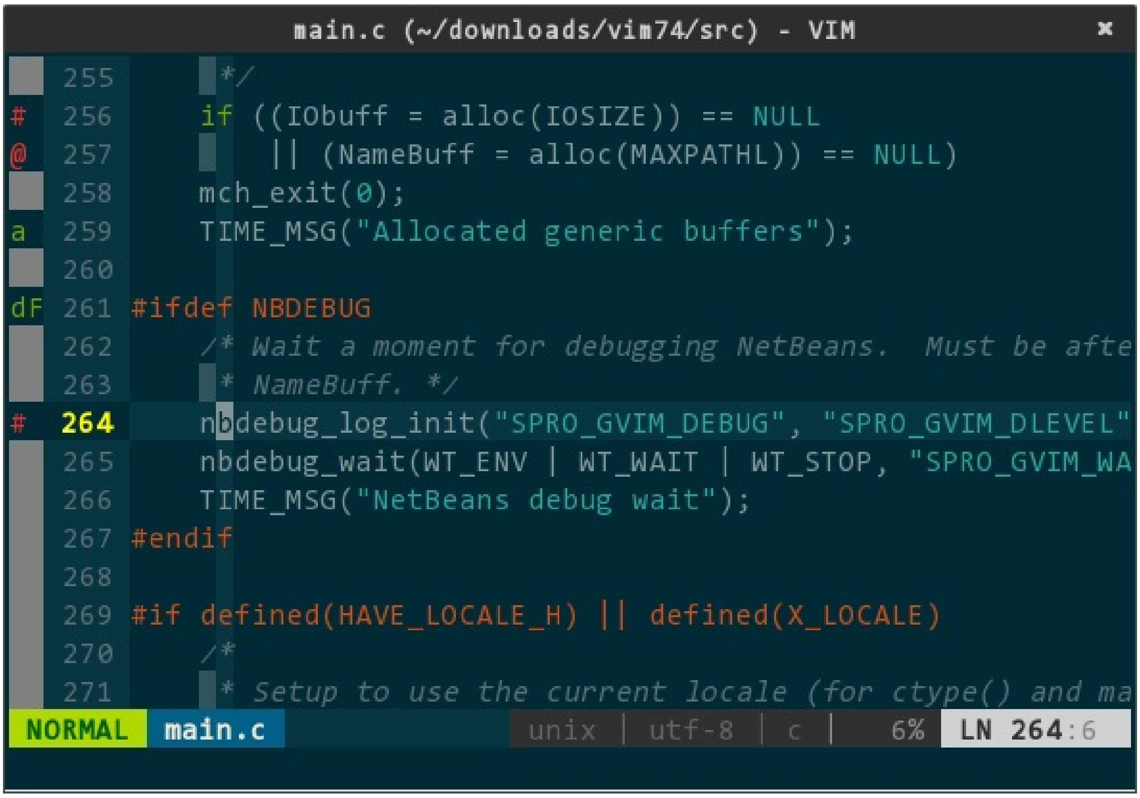
（接口文件与实现文件切换）

上图中，初始状态先打开了接口文件 MyClass.h，键入 ;sw 后，vim 在新 buffer 中打开实现文件 MyClass.cpp，并在当前窗口中显示；再次键入 ;sw 后，当前窗口切回接口文件。

4.5 代码收藏

源码分析过程中，常常需要在不同代码间来回跳转，我需要“收藏”分散在不同处的代码行，以便需要查看时能快速跳转过去，这时，vim 的书签（mark）功能派上大用途了。 vim 书签的使用很简单，在你需要收藏的代码行键入 mm，这样就收藏好了，你试试，没反应？不会吧，难道你 linux 内核编译参数有问题，或者，vim 的编译参数没给全，让我想想，别急，喔，对了，你是指看不到书签？好吧，我承认这是 vim 最大的坑，书签所在行与普通行外观上没任何差别，肉眼，你是找不到他滴。这可不行，得来个让书签可视化的插件，vim-signature（https://github.com/kshenoy/vim-signature ）。vim-signature 通过在书签所在行的前面添加字符的形式，以此可视化书签，这就要求你源码安装的 vim 具备 signs 特性，具体可在 vim 命令模式下键入 ``` :echo has('signs') ``` 若显示 1 则具备该特性，反之 0 则不具备该特性，需参考“1 源码安装编辑器 vim ”重新编译 vim。 vim 的书签分为两类，独立书签和分类书签。独立书签，书签名只能由字母（a-zA-Z）组成，长度最多不超过 2 个字母，并且，同个文件中，不同独立书签名中不能含有相同字母，比如，a 和 bD 可以同时出现在同个文件

在，而 Fc 和 c 则不行。分类书签，书签名只能由可打印特殊字符（!@#$%^&*()）组成，长度只能有 1 个字符，同个文件中，你可以把不同行设置成同名书签，这样，这些行在逻辑上就归类成相同类型的书签了。下图定义了名为 a 和 dF 两个独立书签（分别 259 行和 261 行）、名为 # 的一类分类书签（含 256 行和 264 行）、名为 @ 的一类分类书签（257 行），如下所示：



（独立书签和分类书签）

两种形式的书签完全分布在各自不同的空间中，所以，它俩的任何操作都是互不相同的，比如，你无法遍历所有书签，要么只能在各个独立书签间遍历，要么只能在分类书签间遍历。显然，两种形式的书签都有各自的使用场景，就我而言，只使用独立书签，原因有二：一是独立书签可保存，当我设置好独立书签后关闭文档，下次重新打开该文档时，先前的独立书签仍然有效，而分类书签没有该特性（其他文档环境恢复参见“6.3 环境恢复”）；一是减少记忆快捷键，光是独立书签就有 8 种遍历方式，每种遍历对应一种快捷键，太难记了。

vim-signature 快捷键如下：

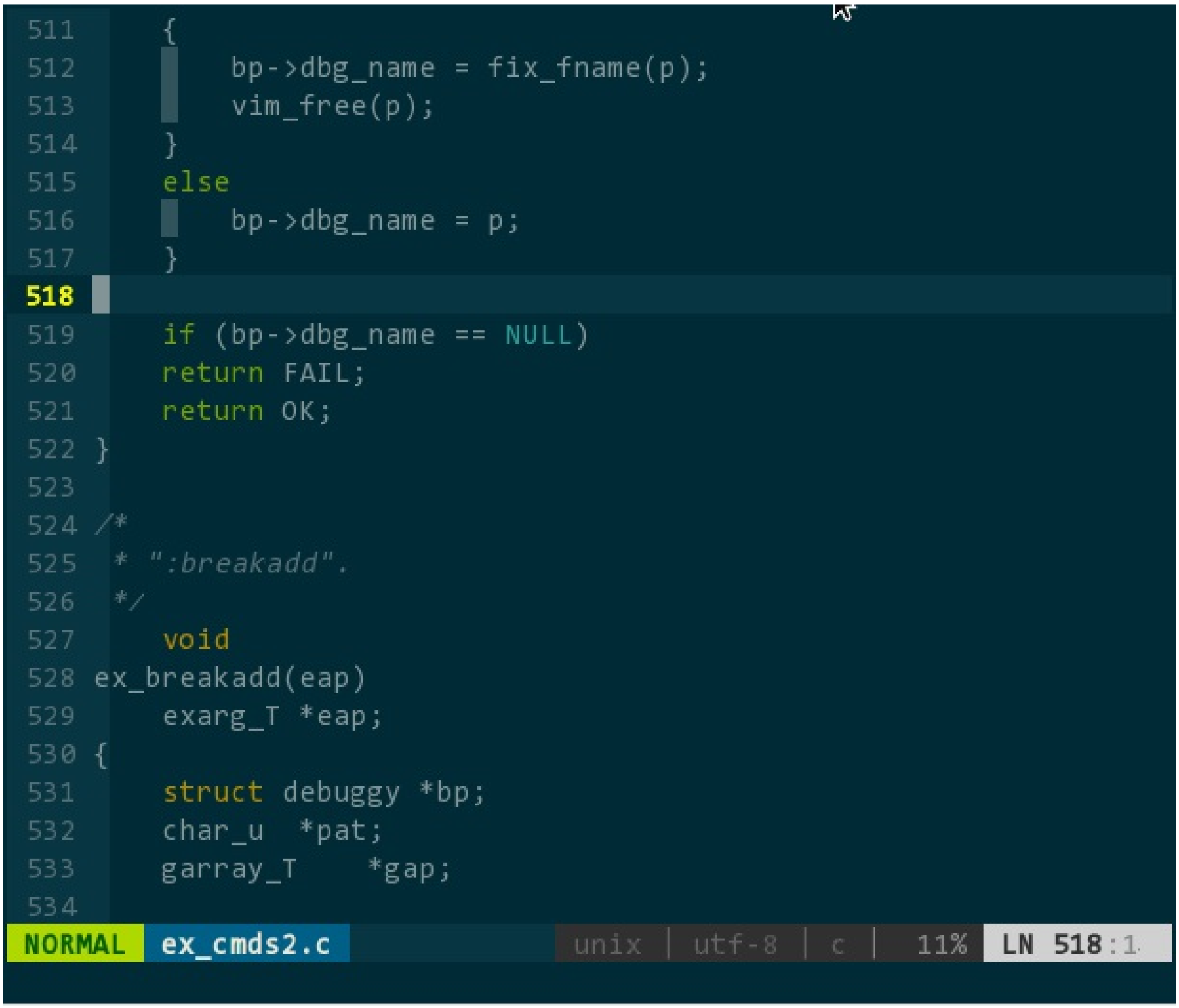
```
let g:SignatureMap = {
        \ 'Leader'             :  "m",
        \ 'PlaceNextMark'      :  "m,",
        \ 'ToggleMarkAtLine'   :  "m.",
        \ 'PurgeMarksAtLine'   :  "m-",
        \ 'DeleteMark'         :  "dm",
        \ 'PurgeMarks'         :  "mda",
        \ 'PurgeMarkers'       :  "m<BS>",
        \ 'GotoNextLineAlpha'  :  "']",
        \ 'GotoPrevLineAlpha'  :  "'[",
        \ 'GotoNextSpotAlpha'  :  "`]",
        \ 'GotoPrevSpotAlpha'  :  "`[",
        \ 'GotoNextLineByPos'  :  "]'",
        \ 'GotoPrevLineByPos'  :  "['",
        \ 'GotoNextSpotByPos'  :  "mn",
        \ 'GotoPrevSpotByPos'  :  "mp",
        \ 'GotoNextMarker'     :  "[+",
        \ 'GotoPrevMarker'     :  "[-",
        \ 'GotoNextMarkerAny'  :  "]=",
        \ 'GotoPrevMarkerAny'  :  "[=",
        \ 'ListLocalMarks'     :  "ms",
        \ 'ListLocalMarkers'   :  "m?"
        \ }
```

够多了吧，粗体部分是按个人习惯重新定义的快捷键，请添加进 .vimrc 中。

常用的操作也就如下几类：

- 书签设定。mx，设定/取消当前行名为 x 的标签；m,，自动设定下一个可用书签名，前面提说，独立书签名是不能重复的，在你已经有了多个独立书签，当想再设置书签时，需要记住已经设定的所有书签名，否则很可能会将已有的书签冲掉，这可不好，所以，vim-signature 为你提供了 m, 快捷键，自动帮你选定下一个可用独立书签名；mda，删除当前文件中所有独立书签。
- 书签罗列。m?，罗列出当前文件中所有书签，选中后回车可直接跳转；
- 书签跳转。mn，按行号前后顺序，跳转至下个独立书签；mp，按行号前后顺序，跳转至前个独立书签。书签跳转方式很多，除了这里说的行号前后顺序，还可以基于书签名字母顺序跳转、分类书签同类跳转、分类书签不同类间跳转等等。

效果如下：



（可视化书签）

我虽然选用了 vim-signature，但不代表它完美了，对我而言，无法在不同文件的书签间跳转绝对算是硬伤。另外，如果觉得收藏的代码行只有行首符号来表示不够醒目，你可以考虑 BOOKMARKS--Mark-and-Highlight-Full-Lines 这个插件（https://github.com/vim-scripts/BOOKMARKS--Mark-and-Highlight-Full-Lines ），它可以让书签行高亮，如下是它的快捷键：，高亮所有书签行；，关闭所有书签行高亮；，清除 [a-z] 的所有书签；，收藏当前行；，取消收藏当前行。

4.6 标识符列表

本节之前的内容，虽说与代码开发有些关系，但最多也只能算作用户体验层面的，真正提升生产效率的内容将从此开始。

本文主题是探讨如何将 vim 打造成高效的 C/C++ 开发环境，希望实现标识符列表、定义跳转、声明提示、实时诊断、代码补全等等系列功能，这些都需要 vim 能够很好地理解我们的代码（不论是 vim 自身还是借助插件甚至第三方工具），如何帮助 vim 理解代码？基本上，有两种主流方式：标签系统和语义系统。至于优劣，简单来说，标签系统配置简单，而语义系统效果精准，后者是趋势。目前对于高阶 IDE 功能，部分已经有对应基于语义的插件支撑，而部分仍只能通过基于标签的方式实现，若同个功能既有语义插件又有标签插件，优选语义。

标签系统

代码中的类、结构、类成员、函数、对象、宏等等这些统称为标识符，每个标识符的定义、所在文件中的行位置、所在文件的路径等等信息就是标签（tag）。

Exuberant Ctags（http://ctags.sourceforge.net/ ，后简称 ctags）就是一款经典的用于生成代码标签信息的工具 。ctags 最初只支持生成 C/C++ 语言，目前已支持 41 种语言，具体列表运行如下命令获取：

```
ctags --list-languages
```

学习知识最好方式就是动手实践。我们以 main.cpp、my_class.h、my_class.cpp 三个文件为例：

第一步，准备代码文件。创建演示目录 /data/workplace/example/、库子目录 /data/workplace/example/lib/，创建如下内容的 main.cpp：

```
#include <iostring>
#include <string>
#include "lib/my_class.h"
using namespace std;
int g_num = 128;
// 重载函数
static void
printMsg (char ch)
{
    std::cout << ch << std::endl;
}
int
main (void)
{
    // 局部对象
    const string    name = "yangyang.gnu";
    // 类
    MyClass    one;
    // 成员函数
    one.printMsg();
    // 使用局部对象
    cout << g_num << name << endl;
    return    (EXIT_SUCCESS);
}
```

创建如下内容的 my_class.h：

```
#pragma once
class MyClass
{
    public:
        void printMsg(void);
    private:
        ;
};
```

创建如下内容的 my_class.cpp：

```
#include "my_class.h"
// 重载函数
static void
printMsg (int i)
{
    std::cout << i << std::endl;
}
void
MyClass::printMsg (void)
{
    std::cout << "I'M MyClass!" << std::endl;
}
```

第二步，生成标签文件。现在运行 ctags 生成标签文件：

```
cd /data/workplace/example/
ctags -R --c++-kinds=+p+l+x+c+d+e+f+g+m+n+s+t+u+v --fields=+liaS --extra=+q --language-fo
```

命令行参数较多，主要关注 --c++-kinds，ctags 默认并不会提取所有标签，运行

```
ctags --list-kinds=c++
```

可看到 ctags 支持生成标签类型的全量列表：

```
c  classes
d  macro definitions
e  enumerators (values inside an enumeration)
f  function definitions
g  enumeration names
l  local variables [off]
m  class, struct, and union members
n  namespaces
p  function prototypes [off]
s  structure names
t  typedefs
u  union names
v  variable definitions
x  external and forward variable declarations [off]
```

其中，标为 off 的局部对象、函数声明、外部对象等类型默认不会生成标签，所以我显式加上所有类型。运行完后，example/ 下多了个文件 tags，内容大致如下：

```
!_TAG_FILE_FORMAT    2    /extended format; --format=1 will not append ;" to lines/
!_TAG_FILE_SORTED    1    /0=unsorted, 1=sorted, 2=foldcase/
!_TAG_PROGRAM_AUTHOR    Darren Hiebert    /dhiebert@users.sourceforge.net/
!_TAG_PROGRAM_NAME    Exuberant Ctags    //
!_TAG_PROGRAM_URL    http://ctags.sourceforge.net    /official site/
!_TAG_PROGRAM_VERSION    5.8    //
MyClass    lib/my_class.h    /^class MyClass $/;"    c
MyClass::printMsg    lib/my_class.cpp    /^MyClass::printMsg (void) $/;"    f    class:My
MyClass::printMsg    lib/my_class.h    /^        void printMsg(void);$/;"    p    class:M
endl    lib/my_class.cpp    /^    std::cout << "I'M MyClass!" << std::endl;$/;"    m    c
endl    lib/my_class.cpp    /^    std::cout << i << std::endl;$/;"    m    class:std    f
endl    main.cpp    /^    cout << g_num << name << endl;$/;"    l
endl    main.cpp    /^    std::cout << ch << std::endl;$/;"    m    class:std    file:
g_num    main.cpp    /^int g_num = 128;$/;"    v
main    main.cpp    /^main (void) $/;"    f    signature:(void)
name    main.cpp    /^    const string    name = "yangyang.gnu";$/;"    l
one    main.cpp    /^    MyClass    one;$/;"    l
printMsg    lib/my_class.cpp    /^MyClass::printMsg (void) $/;"    f    class:MyClass
printMsg    lib/my_class.cpp    /^printMsg (int i) $/;"    f    file:    signature:(int i
printMsg    lib/my_class.h    /^        void printMsg(void);$/;"    p    class:MyClass
printMsg    main.cpp    /^    one.printMsg();$/;"    p    file:    signature:()
printMsg    main.cpp    /^printMsg (char ch) $/;"    f    file:    signature:(char ch)
std::endl    lib/my_class.cpp    /^    std::cout << "I'M MyClass!" << std::endl;$/;"    m
std::endl    lib/my_class.cpp    /^    std::cout << i << std::endl;$/;"    m    class:std
std::endl    main.cpp    /^    std::cout << ch << std::endl;$/;"    m    class:std    fil
```

其中，! 开头的几行是 ctags 生成的软件信息忽略之，下面的就是我们需要的标签，每个标签项至少有如下字段（命令行参数不同标签项的字段数不同）：标识符名、标识符所在的文件名（也是该文件的相对路径）、标识符所在行的内容、标识符类型（如，l 表示局部对象），另外，若是函数，则有函数签名字段，若是成员函数，则有访问属型字段等等。

语义系统

通过 ctags 这类标签系统在一定程度上助力 vim 理解我们的代码，对于 C 语言这类简单语言来说，差不多也够了。近几年，随着 C++11/14 的推出，诸如类型推导、lamda 表达式、模版等等新特性，标签系统显得有心无力，这个星球最了解代码的工具非编译器莫属，如果编译器能在语义这个高度帮助 vim 理解代码，那么我们需要的各项 IDE 功能肯定能达到另一个高度。

语义系统，编译器必不可少。GCC 和 clang 两大主流 C/C++ 编译器，作为语义系统的支撑工具，我选择后者，除了 clang 对新标准支持及时、错误诊断信息清晰这些优点之外，更重要的是，它在高内聚、低耦合方面做得非常好，各类插件可以调用 libclang 获取非常完整的代码分析结果，从而轻松且优雅地实现高阶 IDE 功能。你对语义系统肯定还是比较懵懂，紧接着的“基于语义的声明/定义跳转”会让你有更为直观的了解，现在，请跳转至“7.1 编译器/构建工具集成”，一是了解 clang 相较 GCC 的优势，二是安装好最新版 clang 及其标准库，之后再回来。

基于标签的标识符列表

在阅读代码时，经常分析指定函数实现细节，我希望有个插件能把从当前代码文件中提取出的所有标识符放在一个侧边子窗口中，并且能能按语法规则将标识符进行归类，tagbar（https://github.com/majutsushi/tagbar ）是一款基于标签的标识符列表插件，它自动周期性

调用 ctags 获取标签信息（仅保留在内存，不落地成文件）。安装完 tagbar 后，在 .vimrc 中增加如下信息：

```
" 设置 tagbar 子窗口的位置出现在主编辑区的左边
let tagbar_left=1
" 设置显示／隐藏标签列表子窗口的快捷键。速记：identifier list by tag
nnoremap <Leader>ilt :TagbarToggle<CR>
" 设置标签子窗口的宽度
let tagbar_width=32
" tagbar 子窗口中不显示冗余帮助信息
let g:tagbar_compact=1
" 设置 ctags 对哪些代码标识符生成标签
let g:tagbar_type_cpp = {
    \ 'kinds' : [
         \ 'c:classes:0:1',
         \ 'd:macros:0:1',
         \ 'e:enumerators:0:0',
         \ 'f:functions:0:1',
         \ 'g:enumeration:0:1',
         \ 'l:local:0:1',
         \ 'm:members:0:1',
         \ 'n:namespaces:0:1',
         \ 'p:functions_prototypes:0:1',
         \ 's:structs:0:1',
         \ 't:typedefs:0:1',
         \ 'u:unions:0:1',
         \ 'v:global:0:1',
         \ 'x:external:0:1'
     \ ],
     \ 'sro'        : '::',
     \ 'kind2scope' : {
         \ 'g' : 'enum',
         \ 'n' : 'namespace',
         \ 'c' : 'class',
         \ 's' : 'struct',
         \ 'u' : 'union'
     \ },
     \ 'scope2kind' : {
         \ 'enum'      : 'g',
         \ 'namespace' : 'n',
         \ 'class'     : 'c',
         \ 'struct'    : 's',
         \ 'union'     : 'u'
     \ }
\ }
```

前面提过，ctags 默认并不会提取局部对象、函数声明、外部对象等类型的标签，我必须让 tagbar 告诉 ctags 改变默认参数，这是 tagbar_type_cpp 变量存在的主要目的，所以前面的配置信息中将局部对象、函数声明、外部对象等显式将其加进该变量的 kinds 域中。具体格式为

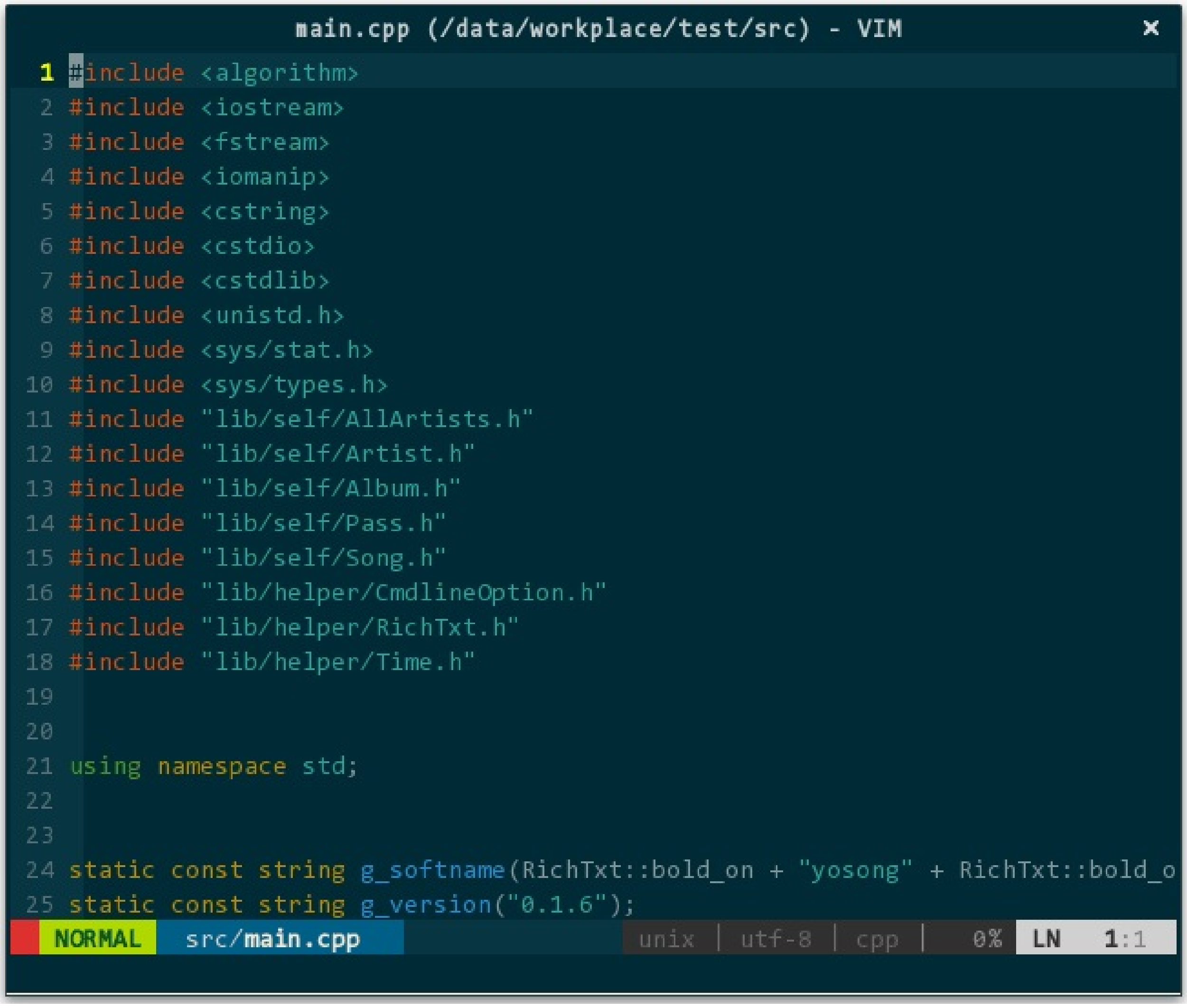
```
{short}:{long}[:{fold}[:{stl}]]
```

用于描述函数、变量、结构体等等不同类型的标识符，每种类型对应一行。其中，short 将作为 ctags 的 --c++-kinds 命令行选项的参数，类似：

```
--c++-kinds=+p+l+x+c+d+e+f+g+m+n+s+t+u+v
```

long 将作为 short 的简要描述展示在 vim 的 tagbar 子窗口中；fold 表示这种类型的标识符是否折叠显示；stl 指定是否在 vim 状态栏中显示附加信息。

重启 vim 后，打开一个 C/C++ 源码文件，键入 ilt，将在左侧的 tagbar 窗口中将可看到标签列表：



（基于标签的标识符列表）

从上图可知 tagbar 的几个特点：* 按作用域归类不同标签。按名字空间 n_foo、类 Foo 进行归类，在内部有声明、有定义；* 显示标签类型。名字空间、类、函数等等；* 显示完整函数原型；* 图形化显示共有成员（+）、私有成员（-）、保护成员（#）；在标识符列表中选中对应标识符后回车即可跳至源码中对应位置；在源码中停顿几秒，tagbar 将高亮对应标识符；每次保存文件时或者切换到不同代码文件时 tagbar 自动调用 ctags 更是标签数据库；tagbar 有两种排序方式，一是按标签名字母先后顺序、一是按标签在源码中出现的先后顺序，在 .vimrc 中我配置选用后者，键入 s 切换不同不同排序方式。

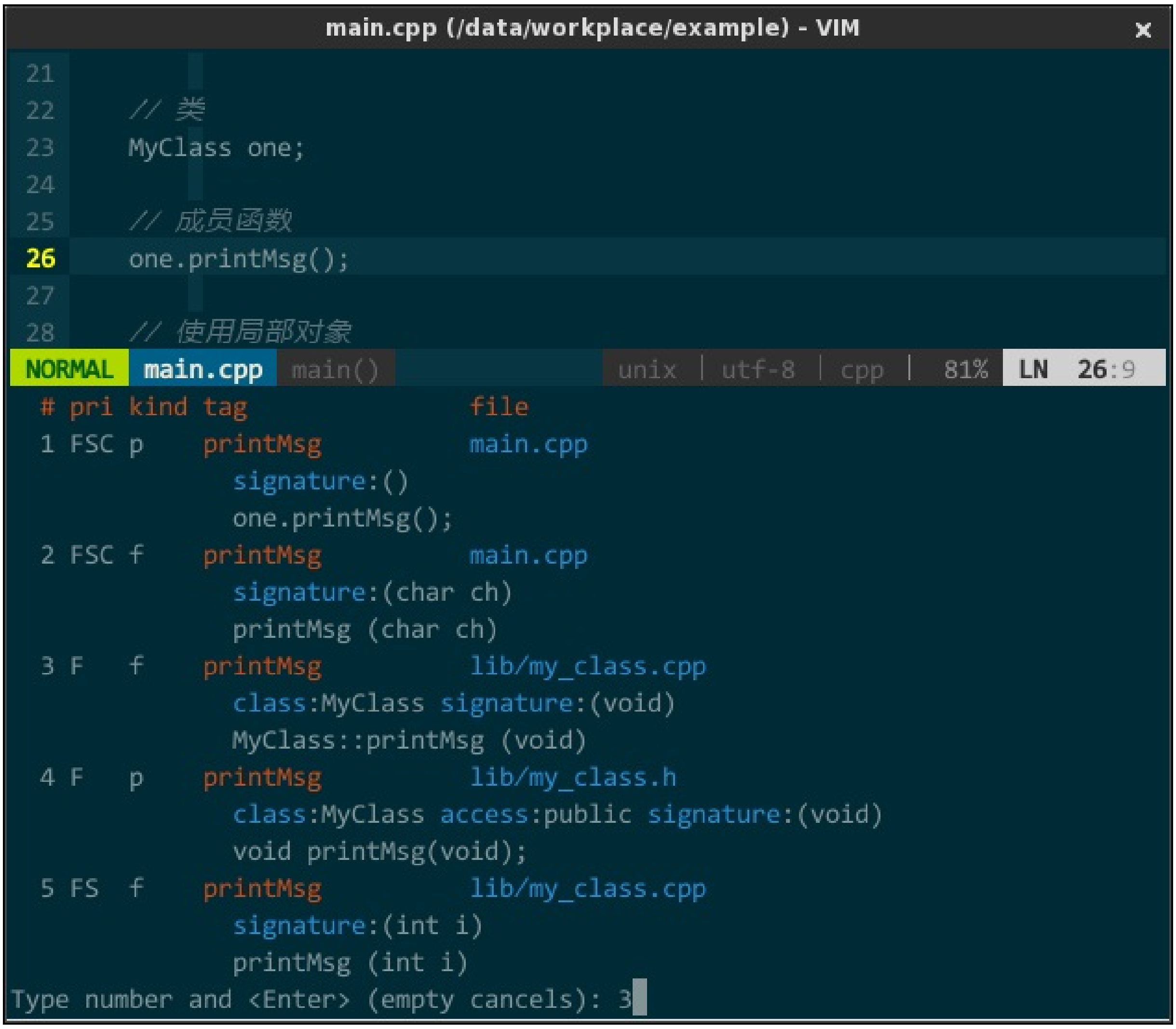
4.7 声明/定义跳转

假设你正在分析某个开源项目源码，在 main.cpp 中遇到调用函数 func()，想要查看它如何实现，一种方式：在 main.cpp 中查找 -> 若没有在工程内查找 -> 找到后打开对应文件 -> 文件

内查找其所在行 -> 移动光标到该行 -> 分析完后切换会先前文件，不仅效率太低更要命的是影响我的思维连续性。我需要另外高效的方式，就像眞正函数调用一样：光标选中调用处的 func() -> 键入某个快捷键自动转换到 func() 实现处 -> 键入某个键又回到 func() 调用处，这就是所谓的定义跳转。 基本上，vim 世界存在两类导航：基于标签的跳转和基于语义的跳转。

基于标签的声明/定义跳转

继续延用前面接收标签系统的例子文件 main.cpp、my_class.h、my_class.cpp，第二步已经生成好了标签文件，那么要实现声明/定义跳转，需要第三步，引入标签文件。这让 vim 知晓标签文件的路径。在 /data/workplace/example/ 目录下用 vim 打开 main.cpp，在 vim 中执行如下目录引入标签文件 tags： ``` :set tags+=/data/workplace/example/tags ``` 既然 vim 有个专门的命令来引入标签，说明 vim 能识别标签。虽然标签文件中并无行号，但已经有标签所在文件，以及标签所在行的完整内容，vim 只需切换至对应文件，再在文件内作内容查找即可找到对应行。换言之，只要有对应的标签文件，vim 就能根据标签跳转至标签定义处。 这时，你可以体验下初级的声明/定义跳转功能。把光标移到 main.cpp 的 one.printMsg() 那行的 printMsg 上，键入快捷键 g]，vim 将罗列出名为 printMsg 的所有标签候选列表，按需选择键入编号即可跳转进入。如下图：



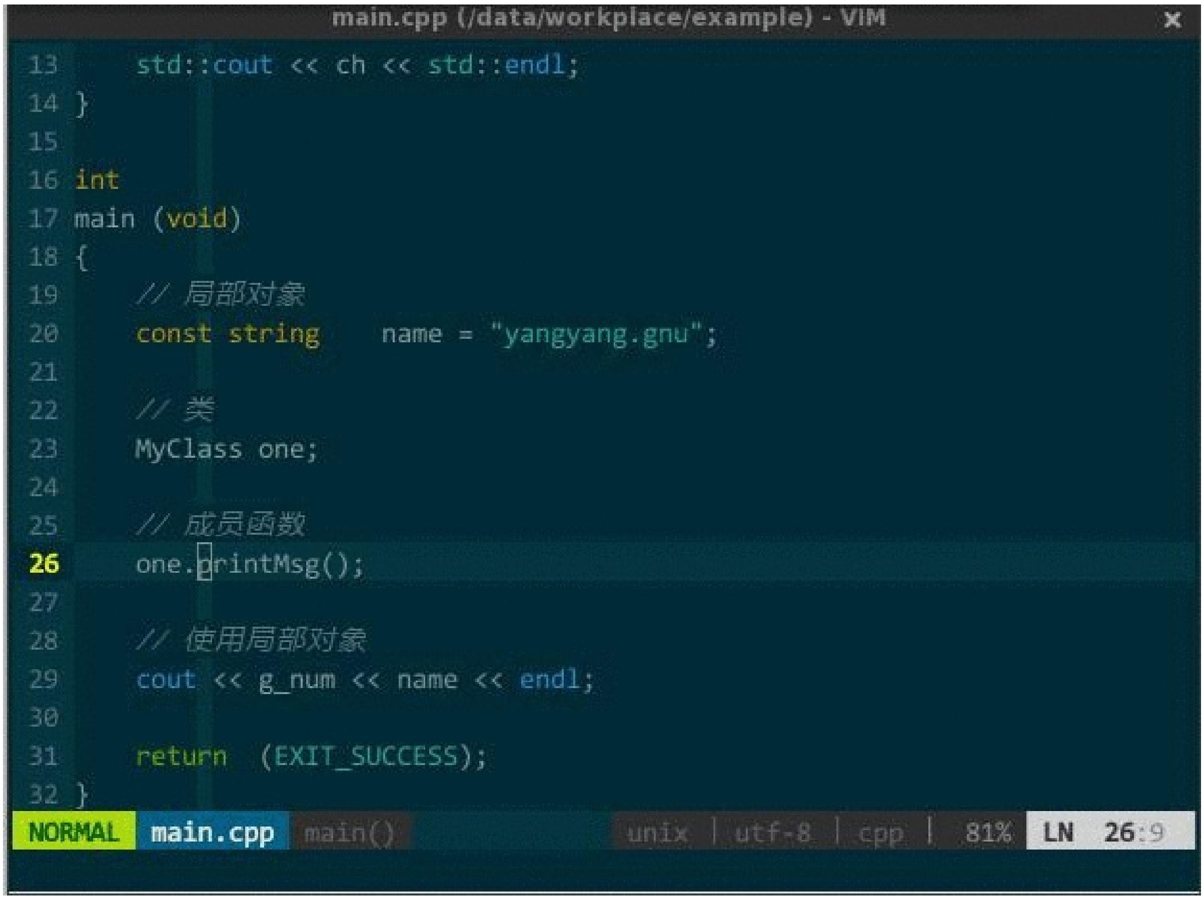
（待选标签）

目前为止，离我预期还有差距。

第一，选择候选列表影响思维连续性。首先得明白为何会出现待选列表。前面说过，vim 做的事情很简单，就是把光标所在单词放到标签文件中查找，如果只有一个，当然你可以直接跳转过去，大部分时候会找到多项匹配标签，比如，函数声明、函数定义、函数调用、函数重载等等都会导致同个函数名出现在多个标签中，vim 无法知道你要查看哪项，只能让你自己选择。其实，因为标签文件中已经包含了函数签名属性，vim 的查找机制如果不是基于关键字，而是基于语义的话，那也可以直接命中，期待后续 vim 有此功能吧。既然无法直接解决，换个思路，我不想选择列表，但可以接受遍历匹配标签。就是说，我不想输入数字选择第几项，但可以接受键入正向快捷键后遍历第一个匹配标签，再次键入快捷键遍历第二个，直到最后一个，键入反向快捷键逆序遍历。这下事情简单了，命令 :tnext 和 :tprevious 分别先后和向前遍历匹配标签，定义两个快捷键搞定：

```
" 正向遍历同名标签
nmap <Leader>tn :tnext<CR>
" 反向遍历同名标签
nmap <Leader>tp :tprevious<CR>
```

等等，这还不行，vim 中有个叫标签栈（tags stack）的机制，:tnext、:tprevious 只能遍历已经压入标签栈内的标签，所以，你在遍历前需要通过快捷键 ctrl-] 将光标所在单词匹配的所有标签压入标签栈中，然后才能遍历。不说复杂了，以后你只需先键入 ctrl-]，若没跳转至需要的标签，再键入 tn 往后或者 tp 往前遍历即可。如下图所示：



（基于标签的跳转）

第二，如何返回先前位置。当分析完函数实现后，我需要返回先前调用处，可以键入 vim 快捷键 ctrl-t 返回，如果想再次进入，可以用前面介绍的方式，或者键入 ctrl-i。另外，注意，ctrl-o 以是一种返回快捷键，但与 ctrl-t 的返回不同，前者是返回上次光标停留行、后者返回上个标签。

第三，如何自动生成标签并引入。开发时代码不停在变更，每次还要手动执行 ctags 命令生成新的标签文件，太麻烦了，得想个法周期性针对这个工程自动生成标签文件，并通知 vim 引人该标签文件，嘿，还真有这样的插件 —— indexer（https://github.com/vim-scripts/indexer.tar.gz ）。indexer 依赖 DfrankUtil（https://github.com/vim-scripts/DfrankUtil ）、vimprj（https://github.com/vim-scripts/vimprj ）两个插件，请一并安装。请在 .vimrc 中增加：

```
" 设置插件 indexer 调用 ctags 的参数
" 默认 --c++-kinds=+p+l，重新设置为 --c++-kinds=+p+l+x+c+d+e+f+g+m+n+s+t+u+v
" 默认 --fields=+iaS 不满足 YCM 要求，需改为 --fields=+iaSl
let g:indexer_ctagsCommandLineOptions="--c++-kinds=+p+l+x+c+d+e+f+g+m+n+s+t+u+v --fields=
```

另外，indexer 还有个自己的配置文件，用于设定各个工程的根目录路径，配置文件位于 ~/.indexer_files，内容可以设定为：

```
--------------- ~/.indexer_files --------------
[foo]
/data/workplace/foo/src/
[bar]
/data/workplace/bar/src/
```

上例设定了两个工程的根目录，方括号内是对应工程名，路径为工程的代码目录，不要包含构建目录、文档目录，以避免将产生非代码文件的标签信息。这样，从以上目录打开任何代码文件时，indexer 便对整个目录创建标签文件，若代码文件有更新，那么在文件保存时，indexer 将自动调用 ctags 更新标签文件，indexer 生成的标签文件以工程名命名，位于 ~/.indexer_files_tags/，并自动引入进 vim 中，那么

```
:set tags+=/data/workplace/example/tags
```

一步也省了。好了，解决了这三个问题后，vim 的代码导航基本已经达到我的预期。

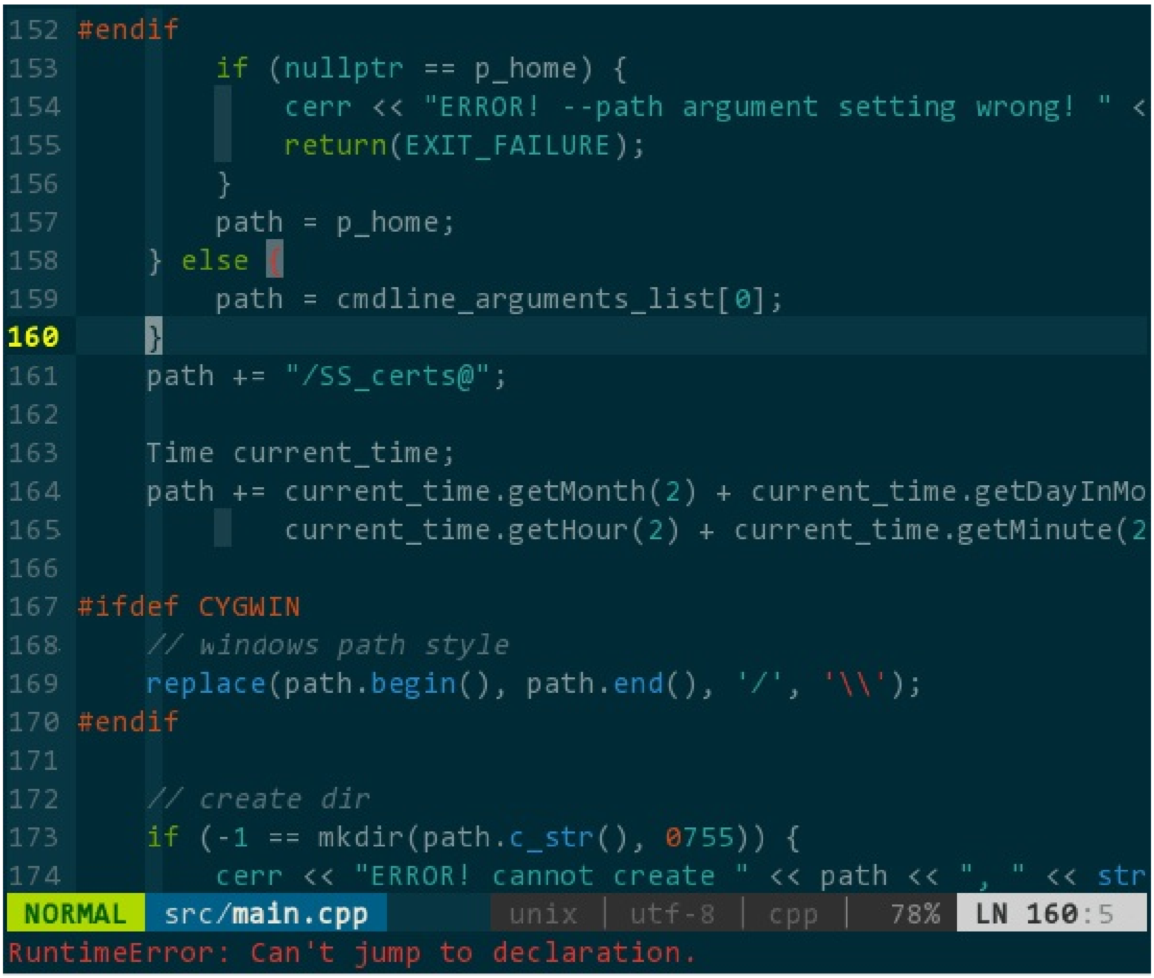
基于语义的声明/定义跳转

有个 vim 插件叫 YCM，有个 C++ 编译器叫 clang，只要正确使用它俩，你将获得无与伦比的代码导航用户体验，以及，代码补全。当然，代价是相对复杂的配置，涉及几个后续章节知识点（“基于语义的智能补全”和“编译器/构建工具集成”），正因如此，此时我只给出快捷键设置，在看完“基于语义的智能补全”后请返回此处，重新查阅。

请增加如下快捷键到 .vimrc 中：

```
nnoremap <leader>jc :YcmCompleter GoToDeclaration<CR>
" 只能是 #include 或已打开的文件
nnoremap <leader>jd :YcmCompleter GoToDefinition<CR>
```

效果如下：



（基于语义的跳转）

另外，基于标签的调整建议你也要了解，有助于你熟悉标签系统，毕竟，使用标签的插件很有几个。

4.8 内容查找

vim 支持正则表达式，那么已经具有强劲的查供能力，在当前文件内查找，vim 的 / 和 ? 查找命令非常好用，但工程内查找，自带的查找用户体验还无法达到我的预期。

内容查找，你第一反应会想到 grep 和 ack 两个工具，没错，它俩强大的正则处理能力无需质疑，如果有插件能在 vim 中集成两个工具之一，那么任何查找任务均可轻松搞定，为此，出现了 grep.vim（https://github.com/yegappan/grep ）和 ack.vim（https://github.com/mileszs/ack.vim ）两个插件，通过它们，你可以在 vim 中自在地使用高度整合的 grep 或 ack 两个外部命令，就像 vim 的内部命令一样：查找时，把光标定位

到待查找关键字上后，通过快捷键立即查找该关键字，查询结果通过列表形式将关键字所在行罗列出来，选择后就能跳转到对应位置。很好，这全部都是我想要的，但是，不是我想要的全部。

你知道，在分析源码时，同个关键字会在不同文件的不同位置多次出现，grep.vim 和 ack.vim 只能“将关键字所在行罗列出来”，如果关键字出现的那几行完全相同，那么，我单凭这个列表是无法确定哪行是我需要的，比如，我查找关键字 cnt，代码中，cnt 在 4 行出现过、64 行、128 行、1024 行都出现过，且每行内容均为

```
++cnt;
```

这时，即便 grep.vim 或 ack.vim 在一个有四个选项的列表中为你罗列出相关行，因为完全相同，所以你也无法确定到底应该查看第几项。换言之，除了罗列关键字所在行之外，我还需要看到所在行的上下几行，这样，有了上下文，我就可以最终决定哪一行是我需要的了。ctrlsf.vim（https://github.com/dyng/ctrlsf.vim ）为此而生。

ctrlsf.vim 后端调用 ack，所以你得提前自行安装，版本不得低于 v2.0，openSUSE 用户可以

```
zypper --no-refresh in ack
```

进行安装。ctrlsf.vim 支持 ack 所有选项，要查找某个关键字（如，yangyang），你可以想让光标定位在该关键字上面，然后命令模式下键入

```
:CtrlSF
```

将自动提取光标所在关键字进行查找，你也可以指定 ack 的选项

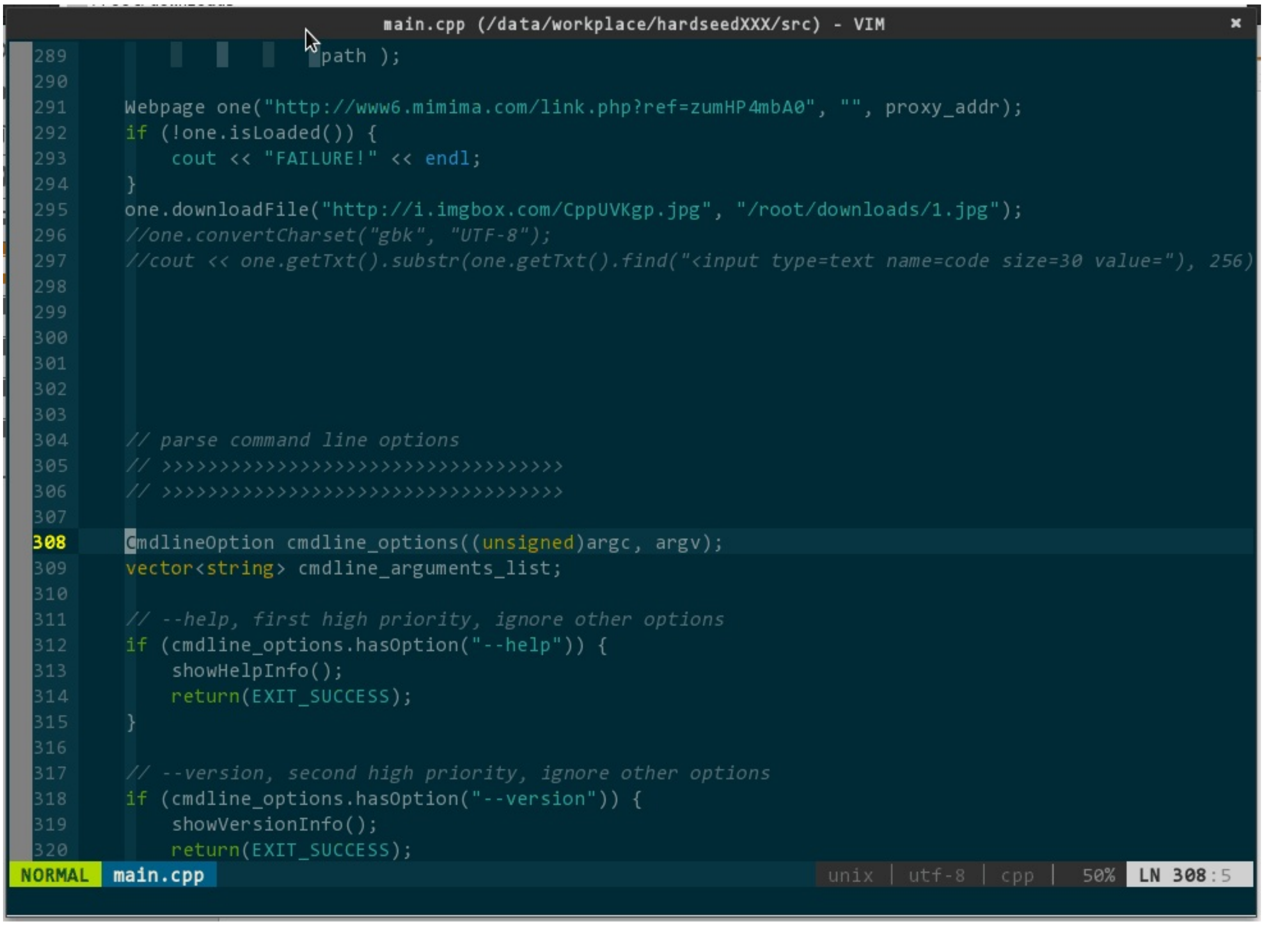
```
:CtrlSF -i -C 1 [pattern] /my/path/
```

为方便操作，我设定了快捷键：

```
" 使用 ctrlsf.vim 插件在工程内全局查找光标所在关键字，设置快捷键。快捷键速记法：search in project
nnoremap <Leader>sp :CtrlSF<CR>
```

避免手工键入命令的麻烦。查找结果将以子窗口在左侧呈现，不仅罗列出所有匹配项，而且给出匹配项的上下文。如果从上下文中你还觉得信息量不够，没事，可以键入 p 键，将在右侧子窗口中给出该匹配项的完整代码，而不再仅有前后几行。不想跳至任何匹配项，可以直接键入 q 退出 ctrlsf.vim；如果有钟意的匹配项，光标定位该项后回车，立即跳至新 buffer 中对应位置。

太性感了，以关键字 CmdlineOption 为例，如下所示：



（内容查找）

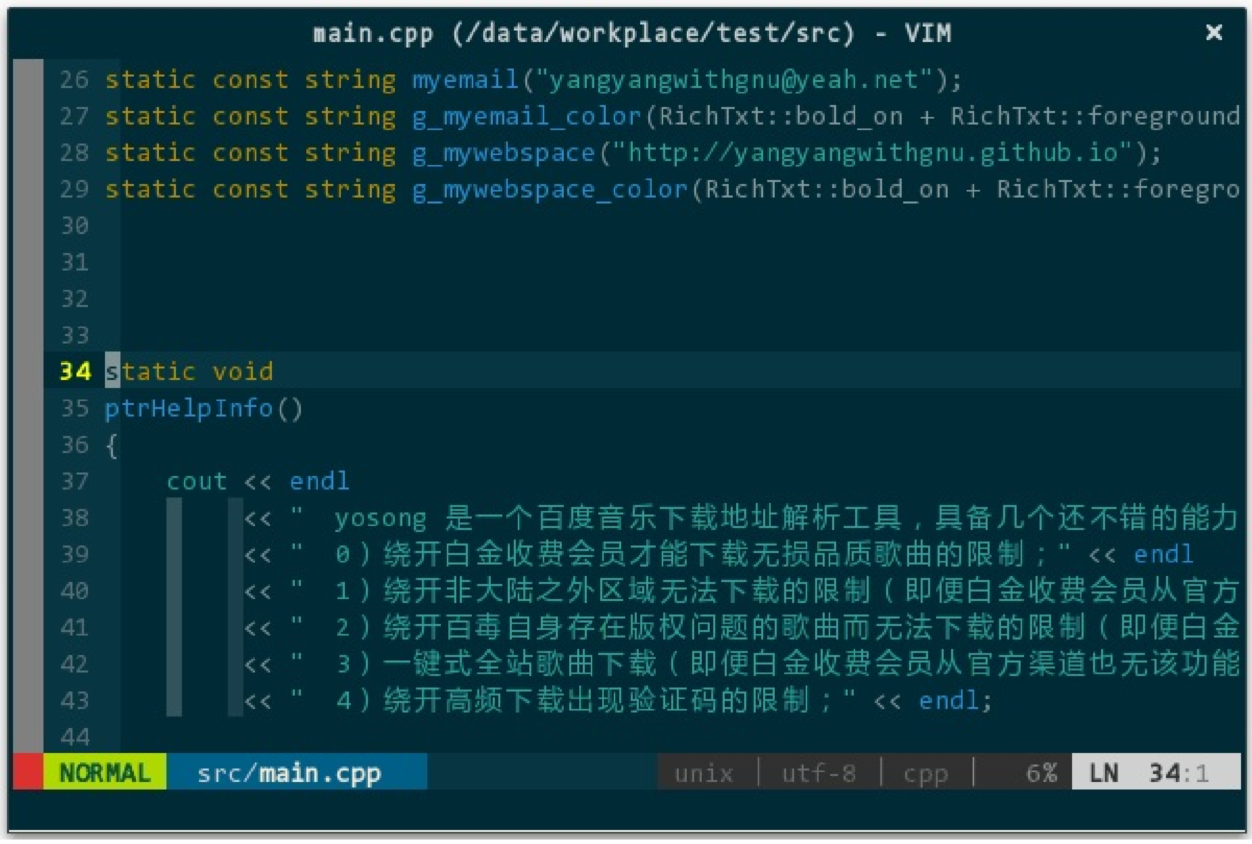
4.9 内容替换

有个名为 iFoo 的全局变量，被工程中 16 个文件引用过，由于你岳母觉得匈牙利命名法严重、异常、绝对以及十分万恶，为讨岳母欢心，不得不将该变量更名为 foo，怎么办？依次打开每个文件，逐一查找后替换？对我而言，内容替换存在两种场景：快捷替换和精确替换。

快捷替换

前面介绍的 ctrlsf 已经把匹配的字符串汇总在侧边子窗口中显示了，同时，它还允许我们直接在该子窗口中进行编辑操作，在这种环境下，如果我们能快捷选中所有匹配字符串，那么就可以先批量删除再在原位插入新的字符串，这岂不是我们需要的替换功能么？

快捷选中 ctrlsf 子窗口中的多个匹配项，关键还是这些匹配项分散在不同行的不同位置，这就需要多光标编辑功能，vim-multiple-cursors 插件（https://github.com/terryma/vim-multiple-cursors ）为次而生。装好 vim-multiple-cursors 后，你随便编辑个文档，随便输入多个相同的字符串，先在可视化模式下选中其中一个，接着键入 ctrl-n，你会发现第二个该字符串也被选中了，持续键入 ctrl-n，你可以选中所有相同的字符串，把这个功能与 ctrlsf 结合，你来感受下：



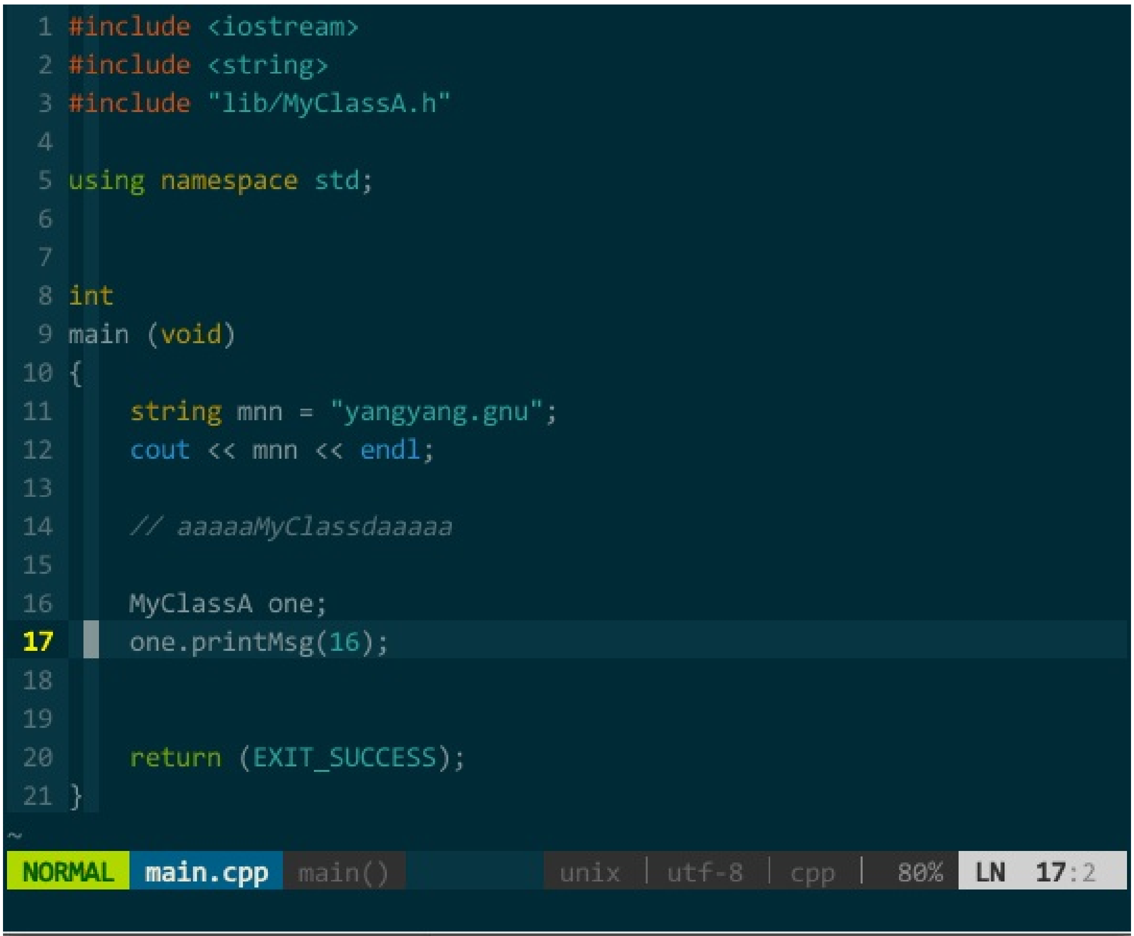
（快捷替换）

上图中，我想将 prtHelpInfo() 更名为 showHelpInfo()，先通过 ctrlsf 找到工程中所有 prtHelpInfo，然后直接在 ctrlsf 子窗口中选中第一个 ptr，再通过 vim-multiple-cursors 选中第二个 ptr，接着统一删除 ptr 并统一键入 show，最后保存并重新加载替换后的文件。 vim-multiple-cursors 默认快捷键与我系统中其他软件的快捷键冲突，按各自习惯重新设置： ``` let g:multi_cursor_next_key='' let g:multi_cursor_skip_key='' ```

精确替换

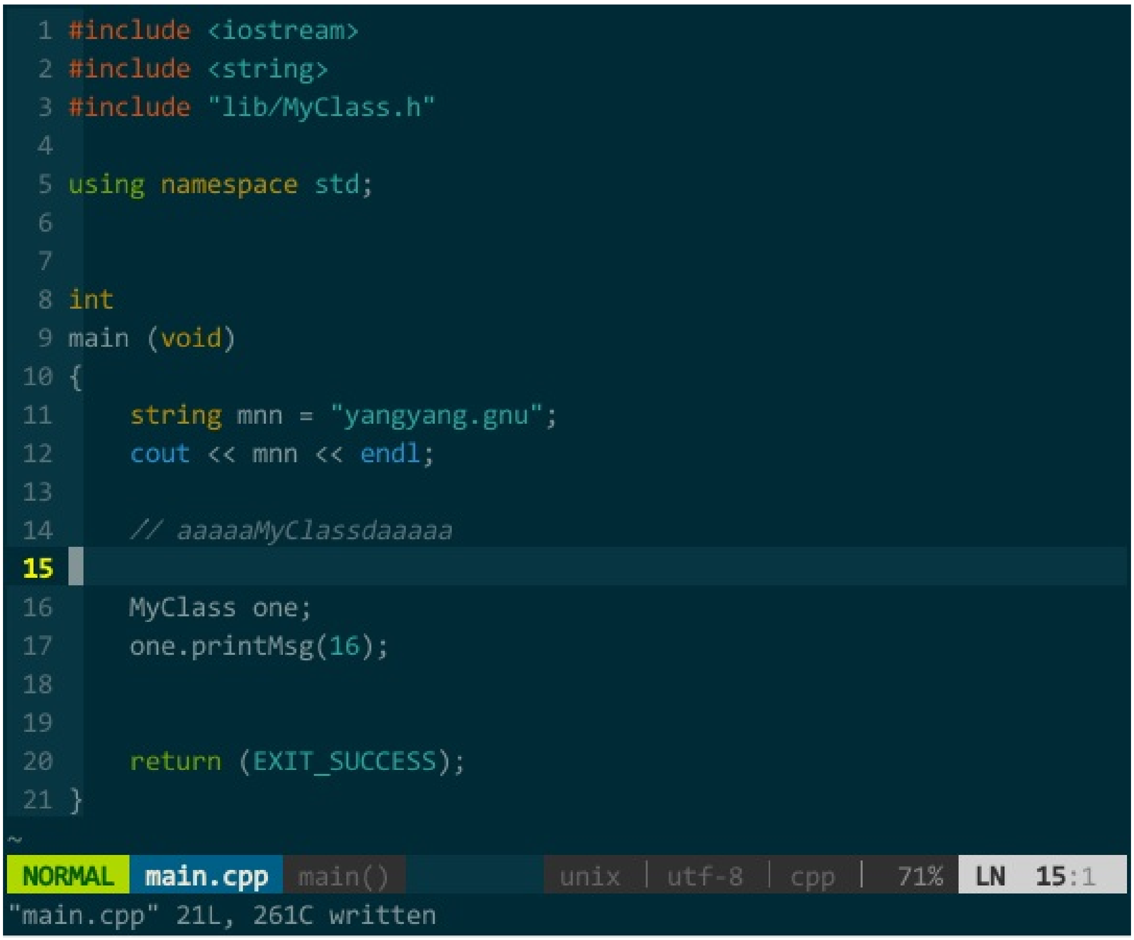
vim 有强大的内容替换命令： ``` :[range]s/{pattern}/{string}/[flags] ``` 在进行内容替换操作时，我关注几个因素：如何指定替换文件范围、是否整词匹配、是否逐一确认后再替换。 如何指定替换文件范围？ * 如果在当前文件内替换，[range] 不用指定，默认就在当前文件内； * 如果在当前选中区域，[range] 也不用指定，在你键入替换命令时，vim 自动将生成如下命令： ``` :'<,'>s/{pattern}/{string}/[flags] ``` * 你也可以指定行范围，如，第三行到第五行： ``` :3,5s/{pattern}/{string}/[flags] ``` * 如果对打开文件进行替换，你需要先通过 :bufdo 命令显式告知 vim 范围，再执行替换； * 如果对工程内所有文件进行替换，先 :args **/*.cpp **/*.h 告知 vim 范围，再执行替换； 是否整词匹配？{pattern} 用于指定匹配模式。如果需要整词匹配，则该字段应由 \< 和 \> 修饰待替换字符串（如，\）；无须整词匹配则不用修饰，直接给定该字符串即可； 是否逐一确认后再替换？[flags] 可用于指定是否需要确认。若无须确认，该字段设定为 ge 即可；有时不见得所有匹配的字符串都需替换，若在每次替换前进行确认，该字段设定为 gec 即可。 是否整词匹配和是否确认两个条件叠加就有 4 种组合：非整词且不确认、非整词且确认、整词且不确认、整词且确认，每次手工输入这些命令真是麻烦；我把这些组合封装到一个函数中，如下 Replace() 所示： ``` " 替换函数。参数说明： " confirm：

是否替换前逐一确认 " wholeword：是否整词匹配 " replace：被替换字符串 function! Replace(confirm, wholeword, replace) wa let flag = '' if a:confirm let flag .= 'gec' else let flag .= 'ge' endif let search = '' if a:wholeword let search .= '\<' .="" escape(expand(''), '/\.*$^~[') . '\>' else let search .= expand('') endif let replace = escape(a:replace, '/\&~') execute 'argdo %s/' . search . '/' . replace . '/' . flag . '| update' endfunction ``` 为最大程度减少手工输入，Replace() 还能自动提取待替换字符串（只要把光标移至待替换字符串上），同时，替换完成后自动为你保存更改的文件。现在要做的就是赋予 confirm、wholeword 不同实参实现 4 种组合，再绑定 4 个快捷键即可。如下：``` " 不确认、非整词 nnoremap R :call Replace(0, 0, input('Replace '.expand('').' with: ')) " 不确认、整词 nnoremap rw :call Replace(0, 1, input('Replace '.expand('').' with: ')) " 确认、非整词 nnoremap rc :call Replace(1, 0, input('Replace '.expand('').' with: ')) " 确认、整词 nnoremap rcw :call Replace(1, 1, input('Replace '.expand('').' with: ')) nnoremap rwc :call Replace(1, 1, input('Replace '.expand('').' with: ')) ``` 我平时用的最多的无须确认但整词匹配的替换模式，即 \rw。请将完整配置信息添加进 .vimrc 中：``` " 替换函数。参数说明： " confirm：是否替换前逐一确认 " wholeword：是否整词匹配 " replace：被替换字符串 function! Replace(confirm, wholeword, replace) wa let flag = '' if a:confirm let flag .= 'gec' else let flag .= 'ge' endif let search = '' if a:wholeword let search .= '\<' .="" escape(expand(''), '/\.*$^~[') . '\>' else let search .= expand('') endif let replace = escape(a:replace, '/\&~') execute 'argdo %s/' . search . '/' . replace . '/' . flag . '| update' endfunction " 不确认、非整词 nnoremap R :call Replace(0, 0, input('Replace '.expand('').' with: ')) " 不确认、整词 nnoremap rw :call Replace(0, 1, input('Replace '.expand('').' with: ')) " 确认、非整词 nnoremap rc :call Replace(1, 0, input('Replace '.expand('').' with: ')) " 确认、整词 nnoremap rcw :call Replace(1, 1, input('Replace '.expand('').' with: ')) nnoremap rwc :call Replace(1, 1, input('Replace '.expand('').' with: ')) ``` 比如，我将工程的所有 *.cpp 和 *.h 中的关键字 MyClassA 按不确认且整词匹配模式替换成 MyClass，所以注释中的关键字不会被替换掉。如下所示：



（不确认且整词匹配模式的替换）

又比如，对当前文件采用需确认且无须整词匹配的模式进行替换，你会看到注释中的关键字也被替换了：



（确认且无须整词匹配模式的替换）

5 代码开发

在具体编码过程中，我需要一系列提高生产力的功能：批量开/关注释、快速输入代码模板、代码智能补全、路径智能补全、从接口生成实现、查看参考库信息等等，我们逐一来实现。

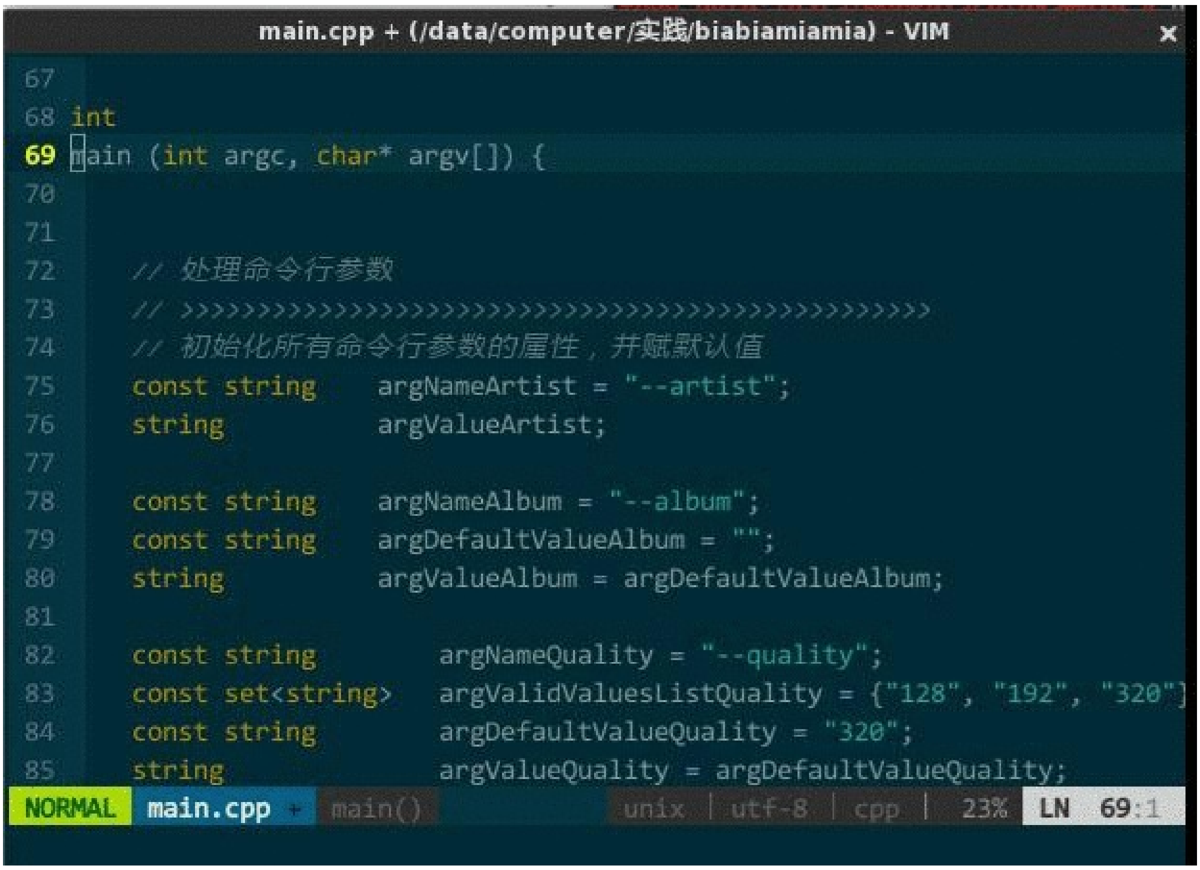
5.1 快速开关注释

需要注释时，到每行代码前输入 //，取消注释时再删除 //，这种方式不是现代人的行为。IDE 应该支持对选中文本块批量（每行）添加注释符号，反之，可批量取消。本来 vim 通过宏方式可以支持该功能，但每次注释时要自己录制宏，关闭 vim 后宏无法保存，所以有人专门编写了一款插件 NERD Commenter（https://github.com/scrooloose/nerdcommenter ），NERD Commenter 根据编辑文档的扩展名自适应采用何种注释风格，如，文档名 x.cpp 则采用 // 注释风格，而 x.c 则是 /**/ 注释风格；另外，如果选中的代码并非整行，那么该插件将用 /**/ 只注释选中部分。

常用操作：

- \cc，注释当前选中文本，如果选中的是整行则在每行首添加 //，如果选中一行的部分内容则在选中部分前后添加分别 /、/；
- \cu，取消选中文本块的注释。

如下图所示：

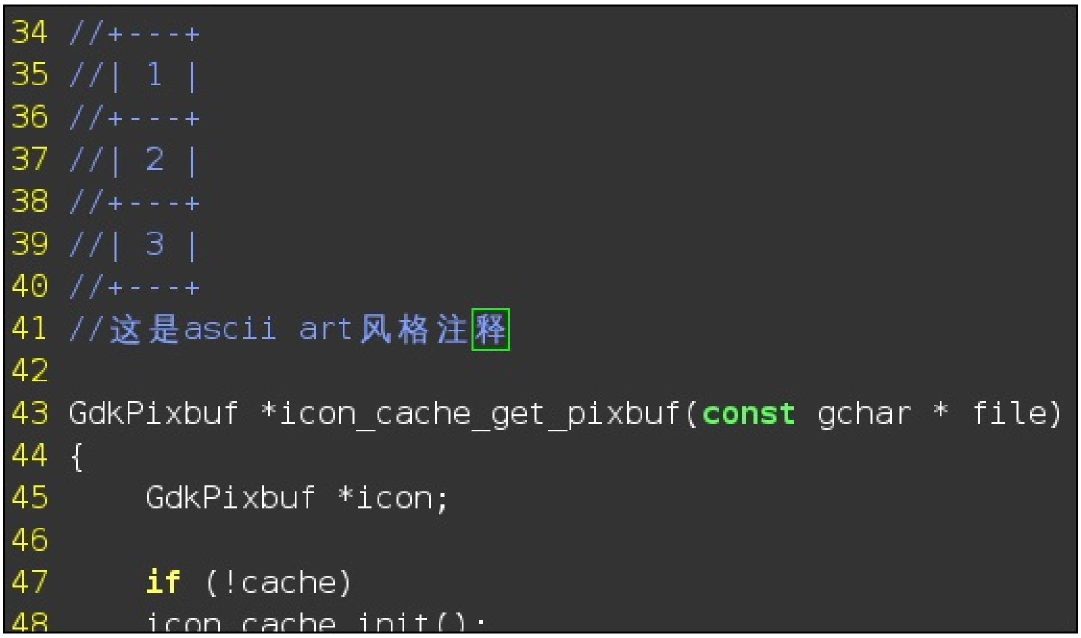


（快速开/关注释）

另外，有时需要 ASCII art 风格的注释，可用 DrawIt!（https://github.com/vim-scripts/DrawIt
），它可以让你用方向键快速绘制出。

常用操作就两个，:Distart，开始绘制，可用方向键绘制线条，空格键绘制或擦除字符；:Distop，停止绘制。

如下图所示：



```
//+---+
//| 1 |
//+---+
//| 2 |
//+---+
//| 3 |
//+---+
//这是ascii art风格注释

GdkPixbuf *icon_cache_get_pixbuf(const gchar * file)
{
    GdkPixbuf *icon;

    if (!cache)
    icon_cache_init();
```

（ASCII art 风格注释）

5.2 模板补全

开发时，我经常要输入相同的代码片断，比如 if-else、switch 语句，如果每个字符全由手工键入，我可吃不了这个苦，我想要简单的键入就能自动帮我完成代码模板的输入，并且光标停留在需要我编辑的位置，比如键入 if，vim 自动完成

```
if (/* condition */) {
    TODO
}
```

而且帮我选中 / *condition* / 部分，不会影响编码连续性 ——
UltiSnips（https://github.com/SirVer/ultisnips ），我的选择。

在进行模板补全时，你是先键入模板名（如，if），接着键入补全快捷键（默认 \），然后 UltiSnips 根据你键入的模板名在代码模板文件中搜索匹配的“模板名-模板”，找到对应模板后，将模板在光标当前位置展开。

UltiSnips 有一套自己的代码模板语法规则，类似：

```
snippet if "if statement" i
if (${1:/* condition */}) {
    ${2:TODO}
}
endsnippet
```

其中，snippet 和 endsnippet 用于表示模板的开始和结束；if 是模板名；"if statement" 是模板描述，你可以把多个模板的模板名定义成一样（如，if () {} 和 if () {} else {} 两模板都定义成相同模板名 if），在模板描述中加以区分（如，分别对应 "if statement" 和 "if-else statement"），这样，在 YCM（重量级智能补全插件） 的补全列表中可以根据模板描述区分选项不同模板；i 是模板控制参数，用于控制模板补全行为，具体参见“快速编辑结对符”一节；${1}、${2} 是 \ 跳转的先后顺序。

新版 UltiSnips 并未自带预定义的代码模板，你可以从 https://github.com/honza/vim-snippets 获取各类语言丰富的代码模板，也可以重新写一套符合自己编码风格的模板。无论哪种方式，你需要在 .vimrc 中设定该模板所在目录名，以便 UltiSnips 寻找到。比如，我自定义的代码模板文件 cpp.snippets，路径为 ~/.vim/bundle/ultisnips/mysnippets/cpp.snippets，对应设置如下： let g:UltiSnipsSnippetDirectories=["mysnippets"] 其中，目录名切勿取为 snippets，这是 UltiSnips 内部保留关键字；另外，目录一定要是 ~/.vim/bundle/ 下的子目录，也就是 vim 的运行时目录。

完整 cpp.snippets 内容如下：

```
#================================
#预处理
#================================
# #include "..."
snippet INC
#include "${1:TODO}"${2}
endsnippet
# #include <...>
snippet inc
#include <${1:TODO}>${2}
endsnippet
#================================
#结构语句
#================================
# if
snippet if
if (${1:/* condition */}) {
    ${2:TODO}
}
endsnippet
# else if
snippet ei
else if (${1:/* condition */}) {
    ${2:TODO}
}
endsnippet
# else
snippet el
else {
    ${1:TODO}
}
endsnippet
# return
snippet re
return(${1:/* condition */});
```

```
endsnippet
# Do While Loop
snippet do
do {
    ${2:TODO}
} while (${1:/* condition */});
endsnippet
# While Loop
snippet wh
while (${1:/* condition */}) {
    ${2:TODO}
}
endsnippet
# switch
snippet sw
switch (${1:/* condition */}) {
    case ${2:c}: {
    }
    break;

    default: {
    }
    break;
}
endsnippet
# 通过迭代器遍历容器（可读写）
snippet for
for (auto ${2:iter} = ${1:c}.begin(); ${3:$2} != $1.end(); ${4:++iter}) {
    ${5:TODO}
}
endsnippet
# 通过迭代器遍历容器（只读）
snippet cfor
for (auto ${2:citer} = ${1:c}.cbegin(); ${3:$2} != $1.cend(); ${4:++citer}) {
    ${5:TODO}
}
endsnippet
# 通过下标遍历容器
snippet For
for (decltype($1.size()) ${2:i} = 0; $2 != ${1}.size(); ${3:++}$2) {
    ${4:TODO}
}
endsnippet
# C++11风格for循环遍历（可读写）
snippet F
for (auto& e : ${1:c}) {
}
endsnippet
# C++11风格for循环遍历（只读）
snippet CF
for (const auto& e : ${1:c}) {
}
endsnippet
# For Loop
snippet FOR
for (unsigned ${2:i} = 0; $2 < ${1:count}; ${3:++}$2) {
    ${4:TODO}
}
endsnippet
# try-catch
snippet try
try {
} catch (${1:/* condition */}) {
}
endsnippet
snippet ca
catch (${1:/* condition */}) {
}
endsnippet
snippet throw
th (${1:/* condition */});
endsnippet
```

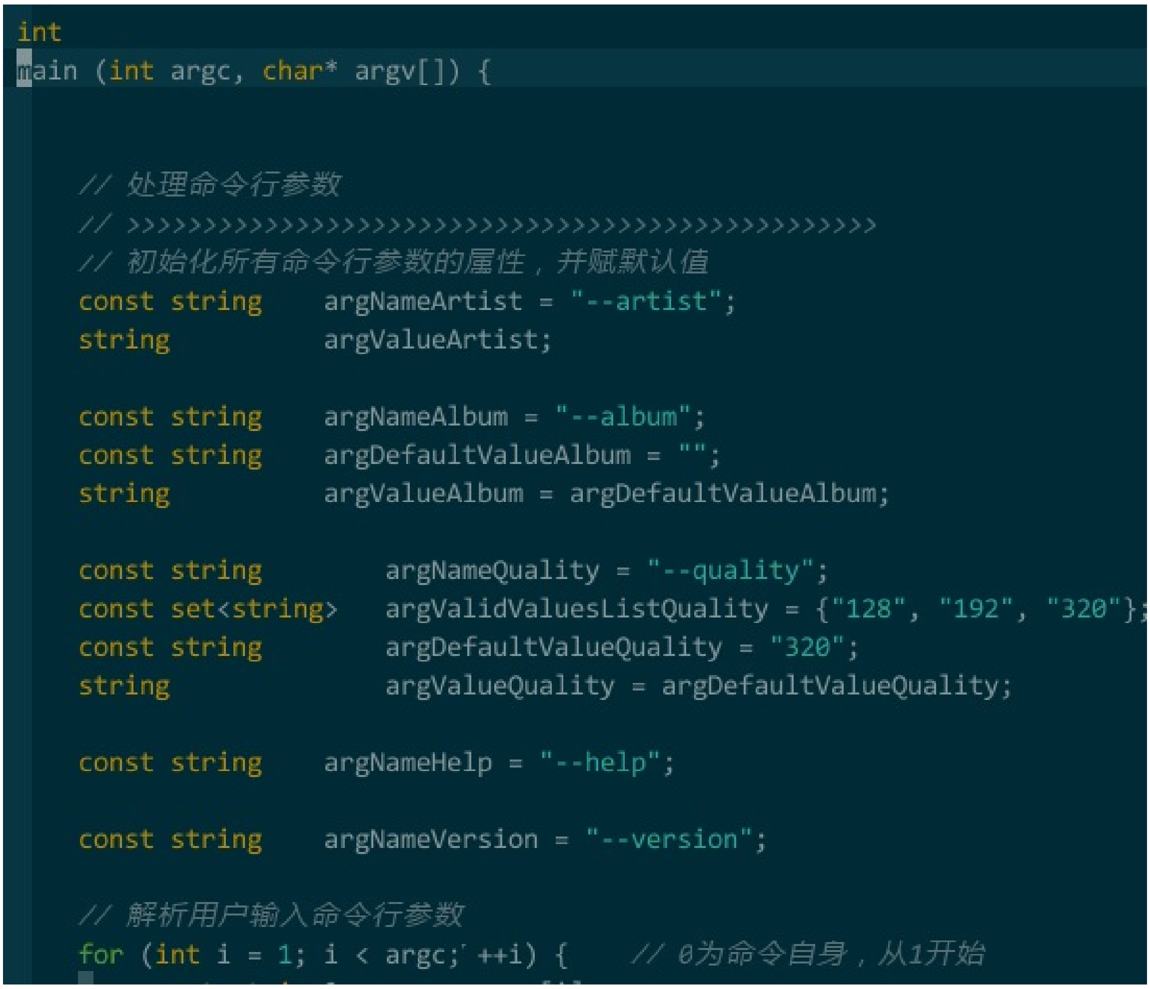
```
#=================================
#容器
#=================================
# std::vector
snippet vec
vector<${1:char}>    v${2};
endsnippet
# std::list
snippet lst
list<${1:char}>    l${2};
endsnippet
# std::set
snippet set
set<${1:key}>    s${2};
endsnippet
# std::map
snippet map
map<${1:key}, ${2:value}>    m${3};
endsnippet
#=================================
#语言扩展
#=================================
# Class
snippet cl
class ${1:`Filename('$1_t', 'name')`}
{
    public:
        $1 ();
        virtual ~$1 ();

    private:
};
endsnippet
#=================================
#结对符
#=================================
 # 括号 bracket
snippet b "bracket" i
(${1})${2}
endsnippet
# 方括号 square bracket，设定为 st 而非 sb，避免与 b 冲突
snippet st "square bracket" i
[${1}]${2}
endsnippet
# 大括号 brace
snippet br "brace" i
{
    ${1}
}${2}
endsnippet
# 单引号 single quote，设定为 se 而非 sq，避免与 q 冲突
snippet se "single quote" I
'${1}'${2}
endsnippet
# 双引号 quote
snippet q "quote" I
"${1}"${2}
endsnippet
# 指针符号 arrow
snippet ar "arrow" i
->${1}
endsnippet
# dot
snippet d "dot" i
.${1}
endsnippet
# 作用域 scope
snippet s "scope" i
::${1}
endsnippet
```

默认情况下，UltiSnips 模板补全快捷键是 \，与后面介绍的 YCM 快捷键有冲突，所有须在 .vimrc 中重新设定：

```
" UltiSnips 的 tab 键与 YCM 冲突，重新设定
let g:UltiSnipsExpandTrigger="<leader><tab>"
let g:UltiSnipsJumpForwardTrigger="<leader><tab>"
let g:UltiSnipsJumpBackwardTrigger="<leader><s-tab>"
```

效果如下：



（模板补全）

5.3 智能补全

真的，这绝对是 G 点。智能补全是提升编码效率的杀手锏。试想下，有个函数叫 getCountAndSizeFromRemotefile()，当你输入 get 后 IDE 自动帮你输入完整的函数名，又如，有个文件 ~/this/is/a/deep/dir/file.txt，就像在 shell 中一样，键入 tab 键自动补全文件路径那是何等惬意！

智能补全有两类实现方式：基于标签的、基于语义的。

基于标签的智能补全

前面代码导航时介绍过标签，每个标签项含有标签名、作用域等等信息，当键入某几个字符时，基于标签的补全插件就在标签文件中搜索匹配的标签项，并罗列出来，你选择中意的，这与前面代码导航类似，一个是用于跳转、一个用于输入。基于标签的补全，后端 ctags 先生成标签文件，前端采用插件 new-omni-completion（内置）进行识别。这种方式操作简单、效果不错，一般来说两步搞定。

第一步，生成标签文件。在工程目录的根目录执行 ctags，该目录下会多出个 tags 文件；

第二步，引入标签文件。在 vim 中引入标签文件，在 vim 中执行命令

```
:set tags+=/home/your_proj/tags
```

后续，在编码时，键入标签的前几个字符后依次键入 ctrl-x ctrl-o 将罗列匹配标签列表、若依次键入 ctrl-x ctrl-i 则文件名补全、ctrl-x ctrl-f 则路径补全。

举个例子，演示如何智能补全 C++ 标准库。与前面介绍的一般步骤一样，先调用 ctags 生成标准库的标签文件，再在 vim 中引入即可，最后编码时由相应插件实时搜索标签文件中的类或模板，显示匹配项：

首先，获取 C++ 标准库源码文件。安装的 GNU C++ 标准库源码文件，openSUSE 可用如下命令：

```
zypper install libstdc++48-devel
```

安装成功后，在 /usr/include/c++/4.8/ 可见到所有源码文件；

接着，执行 ctags 生成标准库的标签文件：

```
cd /usr/include/c++/4.8
ctags -R --c++-kinds=+l+x+p --fields=+iaSl --extra=+q --language-force=c++ -f stdcpp.tags
```

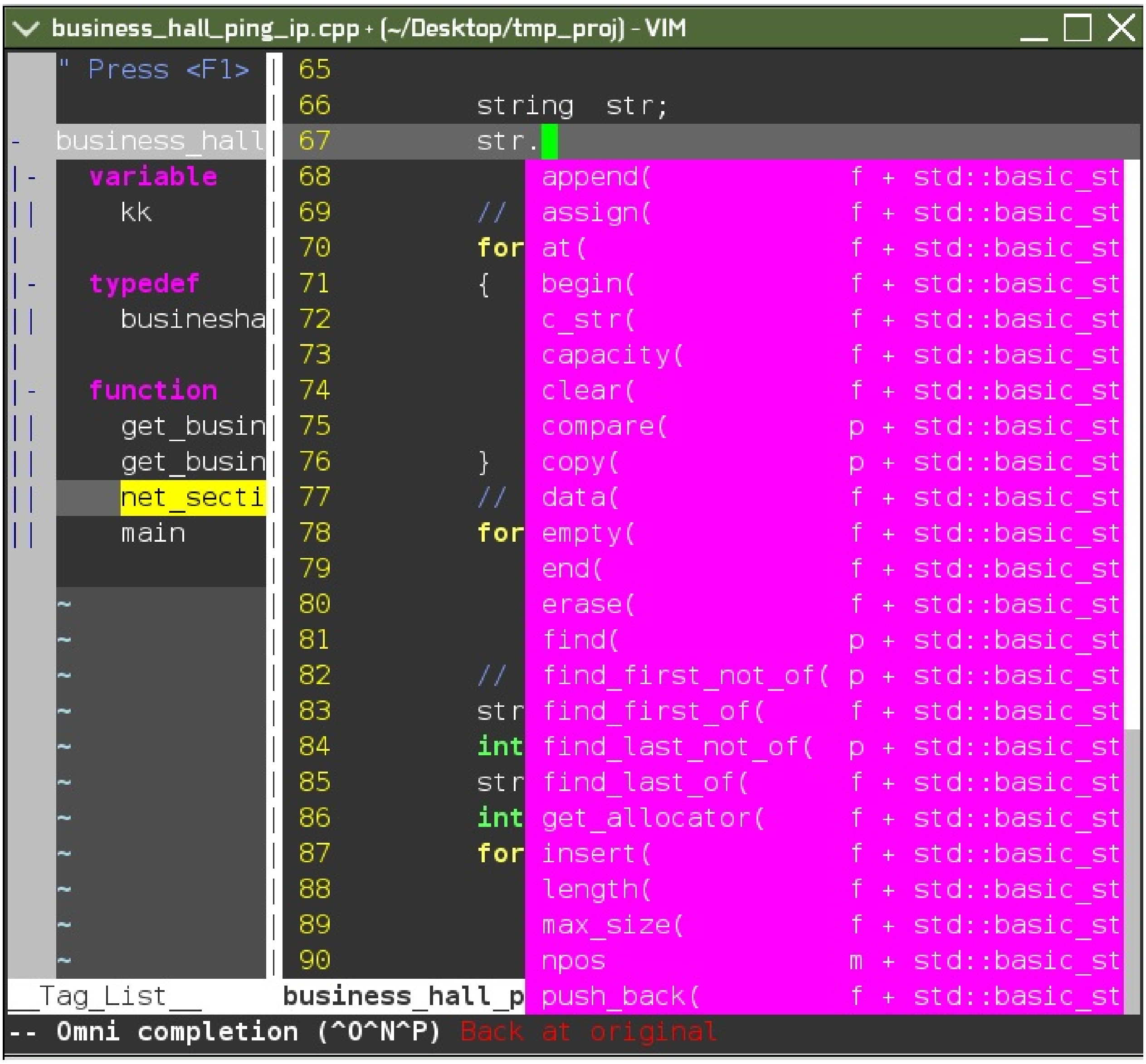
然后，让 OmniCppComplete 成功识别标签文件中的标准库接口。C++ 标准库源码文件中使用了 _GLIBCXX_STD 名字空间（GNU C++ 标准库的实现是这样，如果你使用其他版本的标准库，需要自行查找对应的名字空间名称），标签文件里面的各个标签都嵌套在该名字空间下，所以，要让 OmniCppComplete 正确识别这些标签，必须显式告知 OmniCppComplete 相应的名字空间名称。在.vimrc中增加如下内容：

```
let OmniCpp_DefaultNamespaces = ["_GLIBCXX_STD"]
```

最后，在 vim 中引入该标签文件。在 .vimrc 中增加如下内容：

```
set tags+=/usr/include/c++/4.8/stdcpp.tags
```

后续你就可以进行 C++ 标准库的代码补全，比如，在某个 string 对象名输入 . 时，vim 自动显示成员列表。如下图所示：



（基于标签的 C++ 标准库补全）

没明白？-。-# 咱再来个例子，看看如何补全 linux 系统 API。与前面的标准库补全类似，唯一需要注意，linux 系统 API 头文件中使用了 GCC 编译器扩展语法，必须告诉 ctags 在生成标签时忽略之，否则将生产错误的标签索引。

首先，获取 linux 系统 API 头文件。openSUSE 可用如下命令：

```
zypper install linux-glibc-devel
```

安装成功后，在 /usr/include/ 中可见相关头文件；

接着，执行 ctags 生成系统 API 的标签文件。linux 内核采用 GCC 编译，为提高内核运行效率，linux 源码文件中大量采用 GCC 扩展语法，这影响 ctags 生成正确的标签，必须借由 ctags 的 -I 命令参数告之忽略某些标签，若有多个忽略字符串之间用逗号分割。比如，在文件

unistd.h 中几乎每个API声明中都会出现 **THROW**、nonnull 关键字，前者目的是告诉 GCC 这些函数不会抛异常，尽量多、尽量深地优化这些函数，后者目的告诉 GCC 凡是发现调用这些函数时第一个实参为 nullptr 指针则将其视为语法错误，的确，使用这些扩展语法方便了我们编码，但却影响了 ctags 正常解析，这时可用 -I **THROW,**nonnull 命令行参数让 ctags 忽略这些语法扩展关键字：

```
cd /usr/include/
ctags -R --c-kinds=+l+x+p --fields=+lS -I __THROW,__nonnull -f sys.tags
```

最后，在 vim 中引入该标签文件。在 .vimrc 中增加如下内容：

```
set tags+=/usr/include/sys.tags
```

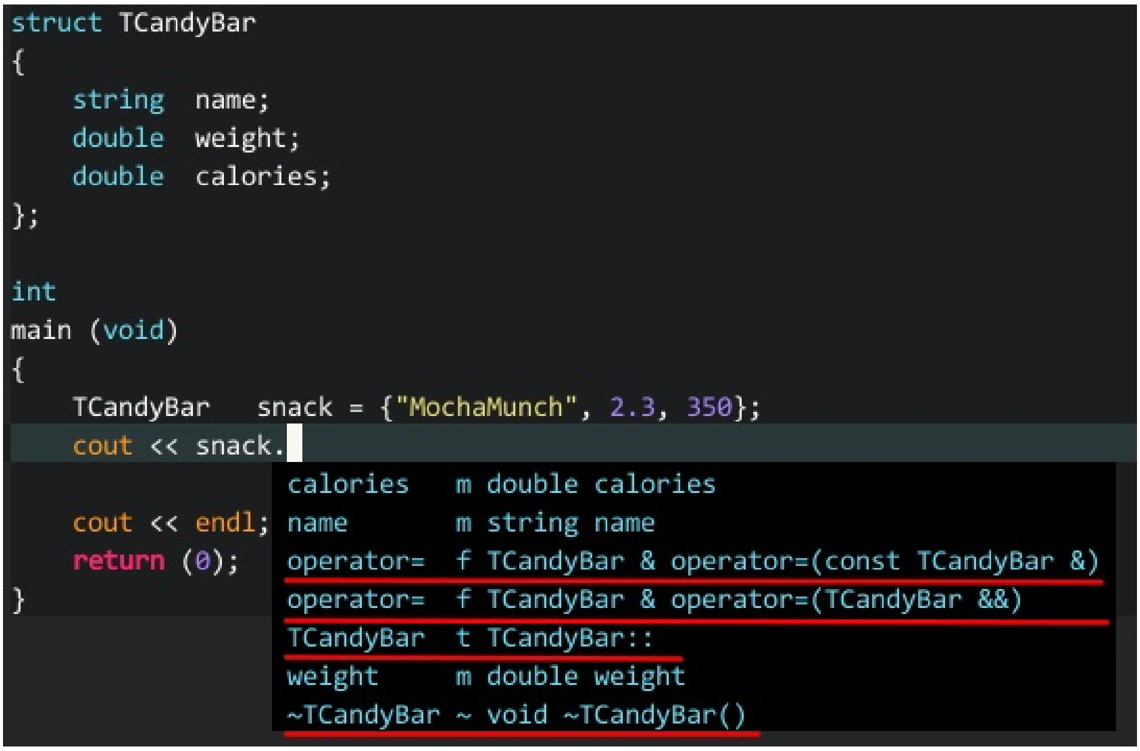
从以上两个例子来看，不论是 C++ 标准库、boost、ACE这些重量级开发库，还是 linux 系统 API 均可遵循"下载源码（至少包括头文件）-执行 ctags 生产标签文件-引入标签文件"的流程实现基于标签的智能补全，若有异常，唯有如下两种可能：一是源码中使用了名字空间，借助 OmniCppComplete 插件的 OmniCpp*DefaultNamespaces* 配置项解决；一是源码中使用了编译器扩展语法，借助 *ctags* 的 *-I* 参数解决（上例仅列举了少量 *GCC* 扩展语法，此外还有 _*attribute_malloc*、__wur 等等大量扩展语法，具体请参见 GCC 手册。以后，如果发现某个系统函数无法自动补全，十有八九是头文件中使用使用了扩展语法，先找到该函数完整声明，再将其使用的扩展语法加入 -I 列表中，最后运行 ctags 重新生产新标签文件即可）。

基于语义的智能补全

对于智能补全只有轻度需求的用户来说，基于标签的补全能够很好地满足需求，但对于我这类重度需求用户来说，但凡涉及标签，就存在以下几个问题：

- 问题一，必须定期调用 ctags 生成标签文件。代码在不停更新，每次智能补全前你得先手动生成标签文件，这还好，你可以借助前面的 indexer 在保存文件时更新标签文件；麻烦的是，你代码中要用到 C++ 标准库中的接口吧，那么事前你先得对整个标准库生成标签文件，这也还好，无非多个步骤；更昏人的是，你得使用了编译器语法扩展的库，你还得一条条找出具体使用了哪些扩展语言，再让 ctags 忽略执行语法关键字，真有够麻烦的；
- 问题二，ctags 本身对 C++ 支持有限。面对函数形参、重载操作符、括号初始化的对象，ctags 有心无力；对于 C++11 新增 lambda 表达式、auto 类型推导更是不认识。

我需要更优的补全机制 —— 基于语义的智能补全。语义补全，实时探测你是否有补全需求，无须你定期生成标签，可解决问题一；语义补全，是借助编译器进行代码分析，只要编译器对 C++ 规范支持度高，不论标准库、类型推导，还是 boost 库中的智能指针都能补全。什么是语义分析补全？看下图：

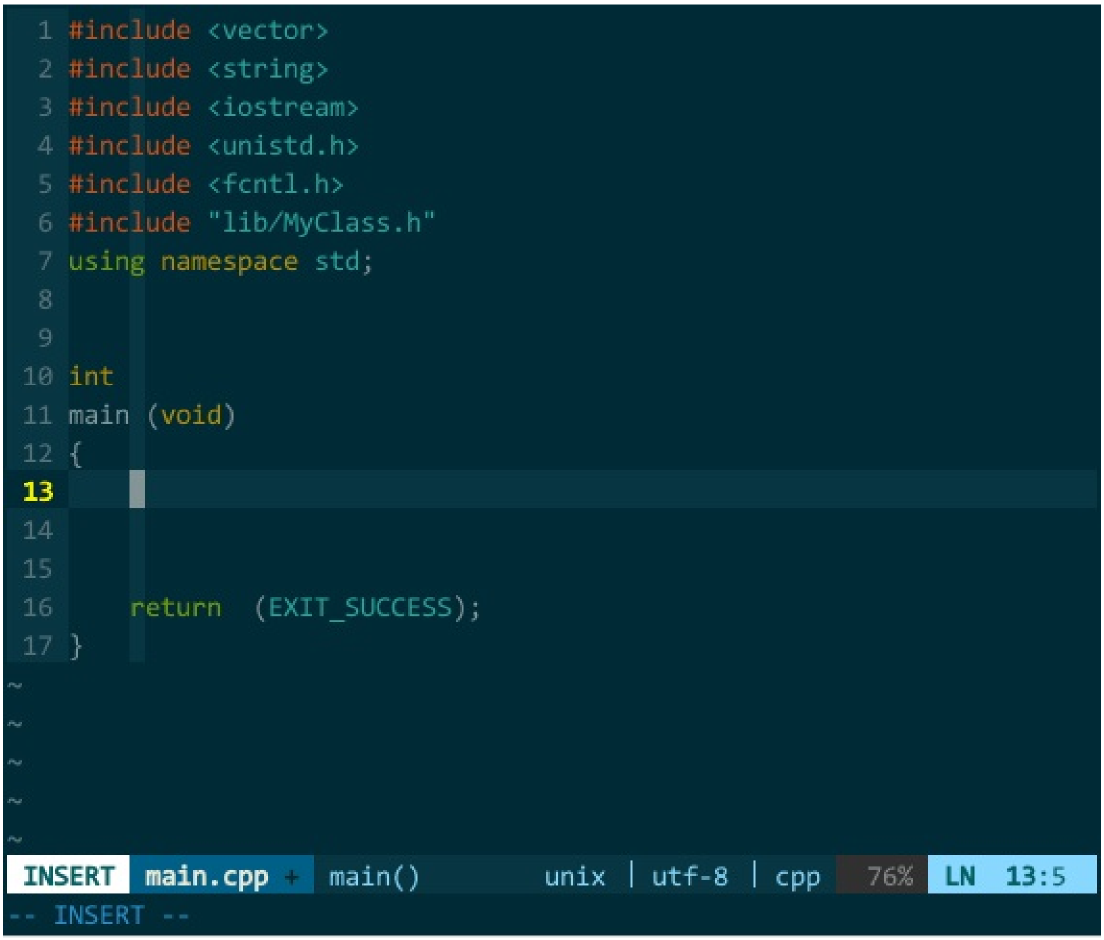


（语义分析补全）

代码中定义的 TCandyBar 类型只包括 3 个成员，但 clang_complete 能补全编译器根据 C++ 规范自动添加的两个重载操作符、一个默认构造函数、一个析构函数，这就是基于语义分析的智能补全。 要进行语义分析，编译器必不可少。linux 上 GCC 和 clang 两大主流 C++ 编译器，基于不同编译器，开源社区分别创造出 GCCSense 和 clang_complete 两个语义补全插件，又得纠结选哪个 -。- ... <穿越> 请跳转至“源码安装编译器 clang”部分做两件事，一是按介绍安装好最新版 clang 及其标准库，二是看明白 clang 相较 GCC 的优势 ... 我选 clang_complete，原因如下： * 使用难度低。clang 采用低耦合设计，语义分析结果（也就是 AST）能以接口形式供外围程序使用，无须任何调整，clang_complete 便能能轻松拿到 clang 输出的语义分析结果；而 GCC 采用高耦合设计，你必须结合补丁重新源码编译 GCC，才能让 GCCSense 接收到它的语义分析结果； * 维护时间长。clang_complete 最新更新为 13 年上，而 GCCSense 则是 09 年下； * 支持跨平台。clang_complete 支持所有平台，而 GCCSense 支持 UNIX-like，不支持 windows。（好啦，这点是我凑数的，我又不用 windows <_<） clang_complete="" 使用简单，在="" vim="" 输入模式下，依次键入要补全的字符，需要弹出补全列表时，手工输入="" \tab。比如有如下代码片断： ``` string name_yang = "yangyang.gnu"; string name_wang = "wangwang"; ``` 我要补全这两个对象，可以先键入 n \tab，这时 clang_complete 将罗列出所有以 n 开头的待选项，接着输入 ame_ 后剩下 name_yang 和 name_wang 两项，按需选择即可。 到这个节目眼上，我应该先给出 clang_complete 的下载地址，再告诉你如何配置 .vimrc，然后给个截图收工，但是、但是，你知道我是个纠结症+完美症重度患者，虽然 clang_complete 相较 ctags+new-omni-completion 的模式有了质的飞跃，仍有雕琢余地： * 无法随键而全。补全动作有感知，要弹出候选列表菜单还得键入 \tab，我希望插件能自动感知我的补全需求； * 无法模糊搜索。上例中，键入的字符要是少了那要出来一堆待选项，选起来眼睛累，多键入几个字符倒是可以让选项少些，但输入多了手又累。这是由于 clang_complete 采用的顺序匹配算法，只要改用子序列匹配算法（模糊搜索算法的一种）即可搞定，这样，我键入 ny 就只出现 name_yang，

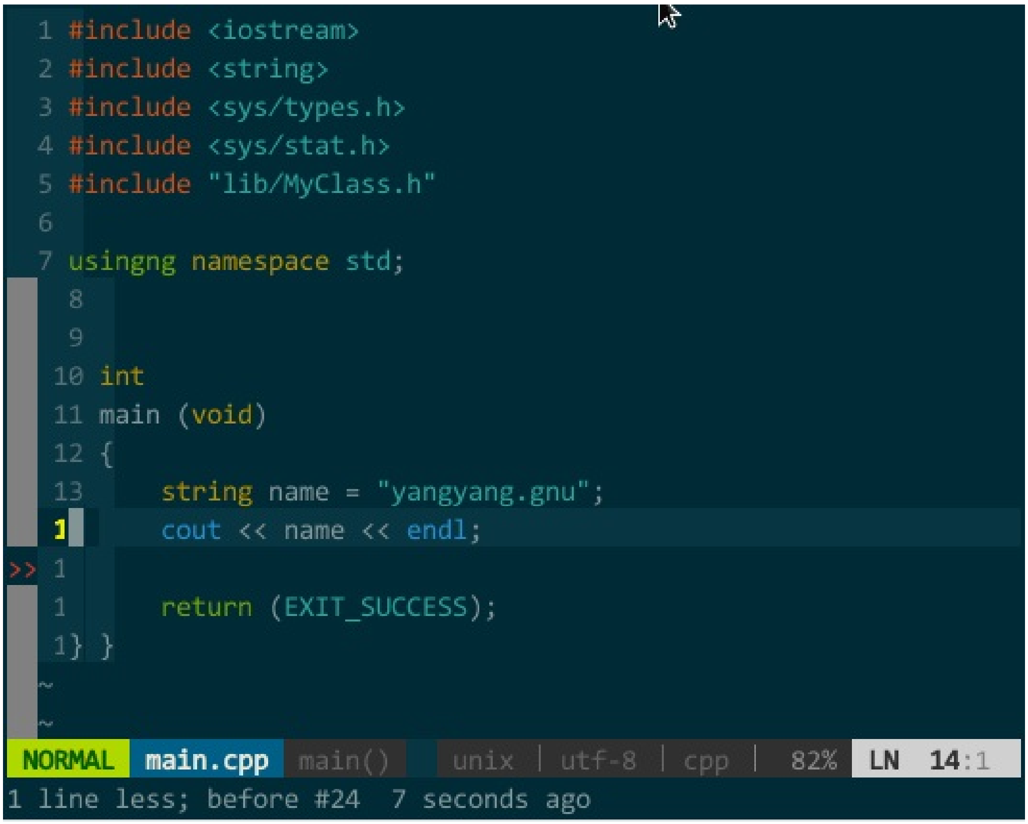
键入 nw 出现 name_wang； * 无法高速补全。补全列表速度不够快，clang_complete 由 python 编写，生成补全列表的速度有一定影响，再加上整个补全动作是在 vim GUI 主线程中执行，所以有时会导致 GUI 假死，我需要由静态语言编写插件内核、动态语言作为粘合剂的补全插件，提升效率； 什么叫所需即所获？就是当你需要什么功能，它就能给你什么功能。 YouCompleteMe（后简称 YCM，https://github.com/Valloric/YouCompleteMe ），一个随键而全的、支持模糊搜索的、高速补全的插件，太棒了！YCM 由 google 公司搜索项目组的软件工程师 Strahinja Val Markovic 所开发，YCM 后端调用 libclang（以获取 AST，当然还有其他语言的语义分析库，我不关注）、前端由 C++ 开发（以提升补全效率）、外层由 python 封装（以成为 vim 插件），它可能是我见过安装最复杂的 vim 插件了。有了 YCM，基本上 clang_complete、AutoComplPop、Supertab、neocomplcache、UltiSnips、 Syntastic 可以再见了。 要运行 YCM，需要 vim 版本至少达到 7.3.598，且支持 python2/3，参照“源码安装编辑器 vim”部分可满足； YCM 含有与 vim 交互的插件部分，以及与 libclang 交互的自身共享库两部分，整个安装过程如下： 第一步，通过 vundle 安装 YCM 插件： ``` Plugin 'Valloric/YouCompleteMe' ``` 随后进入 vim 执行 :PluginInstall。 第二步，下载 libclang。你系统中可能已有现成的 libclang（自行源码编译或者发行套件中自带的），最好别用，YCM 作者强烈建议你下载 LLVM 官网的提供预编译二进制文件，以避免各种妖人问题。在 http://llvm.org/releases/download.html 找到最新版 LLVM，Pre-built Binaries 下选择适合你发行套件的最新版预编译二进制文件，下载并解压至 ~/downloads/clang+llvm； 第三步，编译 YCM 共享库： ``` zypper --no-refresh se python-devel python3-devel boost-devel llvm-clang-devel cd ~/downloads/ mkdir ycm_build cd ycm_build cmake -G "Unix Makefiles" -DUSE_SYSTEM_BOOST=ON -DPATH_TO_LLVM_ROOT=~/downloads/clang+llvm/ .\ ~/.vim/bundle/YouCompleteMe/third_party/ycmd/cpp cmake --build . --target ycm_core ``` 在 ~/.vim/bundle/YouCompleteMe/third_party/ycmd 中将生成 ycm_client_support.so、ycm_core.so、libclang.so 等三个共享库文件； 按照惯例，我该介绍 YCM 的设置。 设置一，YCM 后端调用 libclang 进行语义分析，而 libclang 有很多参数选项（如，是否支持 C++11 的 -std=c++11、是否把警告视为错误的 -Werror），必须有个渠道让 YCM 能告知 libclang，这可以在 .vimrc 中增加一个全局配置，但我有多个工程，每个工程使用的 libclang 参数选项不同岂不是每次都要调整 .vimrc？！YCM 采用更具弹性的方式，每个工程有一个名为 .ycm_extra_conf.py 的私有配置文件，在此文件中写入工程的编译参数选项。下面是个完整的例子： ``` import os import ycm_core flags = ['-std=c++11', '-O0', '-Werror', '-Weverything', '-Wno-documentation', '-Wno-deprecated-declarations', '-Wno-disabled-macro-expansion', '-Wno-float-equal', '-Wno-c++98-compat', '-Wno-c++98-compat-pedantic', '-Wno-global-constructors', '-Wno-exit-time-destructors', '-Wno-missing-prototypes', '-Wno-padded', '-Wno-old-style-cast', '-Wno-weak-vtables', '-x', 'c++', '-I', '.', '-isystem', '/usr/include/',] compilation_database_folder = '' if compilation_database_folder: database = ycm_core.CompilationDatabase(compilation_database_folder) else: database = None SOURCE_EXTENSIONS = ['.cpp', '.cxx', '.cc', '.c', '.m', '.mm'] def DirectoryOfThisScript(): return os.path.dirname(os.path.abspath(__file__)) def MakeRelativePathsInFlagsAbsolute(flags, working_directory): if not working_directory: return list(flags) new_flags = [] make_next_absolute = False path_flags = ['-isystem', '-I', '-

iquote', '--sysroot='] for flag in flags: new_flag = flag if make_next_absolute: make_next_absolute = False if not flag.startswith('/'): new_flag = os.path.join(working_directory, flag) for path_flag in path_flags: if flag == path_flag: make_next_absolute = True break if flag.startswith(path_flag): path = flag[len(path_flag):] new_flag = path_flag + os.path.join(working_directory, path) break if new_flag: new_flags.append(new_flag) return new_flags def IsHeaderFile(filename): extension = os.path.splitext(filename)[1] return extension in ['.h', '.hxx', '.hpp', '.hh'] def GetCompilationInfoForFile(filename): if IsHeaderFile(filename): basename = os.path.splitext(filename)[0] for extension in SOURCE_EXTENSIONS: replacement_file = basename + extension if os.path.exists(replacement_file): compilation_info = database.GetCompilationInfoForFile(replacement_file) if compilation_info.compiler_flags_: return compilation_info return None return database.GetCompilationInfoForFile(filename) def FlagsForFile(filename, **kwargs): if database: compilation_info = GetCompilationInfoForFile(filename) if not compilation_info: return None final_flags = MakeRelativePathsInFlagsAbsolute(compilation_info.compiler_flags_, compilation_info.compiler_working_dir_) else: relative_to = DirectoryOfThisScript() final_flags = MakeRelativePathsInFlagsAbsolute(flags, relative_to) return { 'flags': final_flags, 'do_cache': True } ``` 基本上，根据你工程情况只需调整 .ycm_extra_conf.py 的 flags 部分，前面说过， flags 用于 YCM 调用 libclang 时指定的参数，通常应与构建脚本保持一致（如，CMakeLists.txt）。flags 会产生两方面影响，一是影响 YCM 的补全内容、一是影响代码静态分析插件 syntastic 的显示结果（详见后文“静态分析器集成”）。另外，你得注意，该配置文件其实就是个 python 脚本，python 把缩进视为语法，如果你是直接拷贝文中的 .ycm_extra_conf.py 小心缩进部分。 设置二，在 .vimrc 中增加如下配置信息： ``` " YCM 补全菜单配色 " 菜单 highlight Pmenu ctermfg=2 ctermbg=3 guifg=#005f87 guibg=#EEE8D5 " 选中项 highlight PmenuSel ctermfg=2 ctermbg=3 guifg=#AFD700 guibg=#106900 " 补全功能在注释中同样有效 let g:ycm_complete_in_comments=1 " 允许 vim 加载 .ycm_extra_conf.py 文件，不再提示 let g:ycm_confirm_extra_conf=0 " 开启 YCM 标签补全引擎 let g:ycm_collect_identifiers_from_tags_files=1 " 引入 C++ 标准库tags set tags+=/data/misc/software/misc./vim/stdcpp.tags " YCM 集成 OmniCppComplete 补全引擎，设置其快捷键 inoremap ; " 补全内容不以分割子窗口形式出现，只显示补全列表 set completeopt-=preview " 从第一个键入字符就开始罗列匹配项 let g:ycm_min_num_of_chars_for_completion=1 " 禁止缓存匹配项，每次都重新生成匹配项 let g:ycm_cache_omnifunc=0 " 语法关键字补全 let g:ycm_seed_identifiers_with_syntax=1 ``` 其中大部分内容从注释就能了解，粗体配置项见下文。 YCM 集成了多种补全引擎：语义补全引擎、标签补全引擎、OmniCppComplete 补全引擎、其他补全引擎。 YCM 的语义补全。YCM 不会在每次键入事件上触发语义补全，YCM 作者认为这会影响补全效率而且没什么必要（我持保留意见），YCM 只在如下两种场景下触发语义补全：一是补全标识符所在文件必须在 buffer 中（即，文件已打开）；一是在对象后键入 .、指针后键入 ->、名字空间后键入 ::。如下图所示：



（YCM 的语义补全）

上图中，我先尝试补全类 MyClass 失败，接着我把 MyClass 所在的文件 MyClass.h 打开后切回 main.cpp 再次补全类 MyClass 成功，然后在对象 mc 后键入 . 进行成员补全； YCM 的标签补全。语义补全的确强大，但受限挺多，如果我要补全 STL 中的泛型算法 count_if() 岂不是还要先打开库头文件 algorithm？不用，YCM 也支持标签补全。要使用标签补全，你需要做两件事：一是让 YCM 启用标签补全引擎、二是引入 tag 文件，具体设置如下： ``` " 开启 YCM 标签引擎 let g:ycm_collect_identifiers_from_tags_files=1 " 引入 C++ 标准库tags set tags+=/data/misc/software/misc./vim/stdcpp.tags ``` 其中，工程自身代码的标签可借助 indexer 插件自动生成自动引入，但由于 YCM 要求 tag 文件中必须含有 language:\ 字段（ctags 的命令行参数 --fields 必须含有 l 选项），所有必须通过 indexer_ctagsCommandLineOptions 告知 indexer 调用 ctags 时注意生成该字段，具体设置参见“代码导航”章节；前面章节介绍过如何生成、引入 C++ 标准库的 tag 文件，设置成正确路径即可。另外，由于引入过多 tag 文件会导致 vim 变得非常缓慢，我的经验是，只引入工程自身（indexer 自动引入）和 C++ 标准库的标签（上面配置的最后一行）。如下图所示：

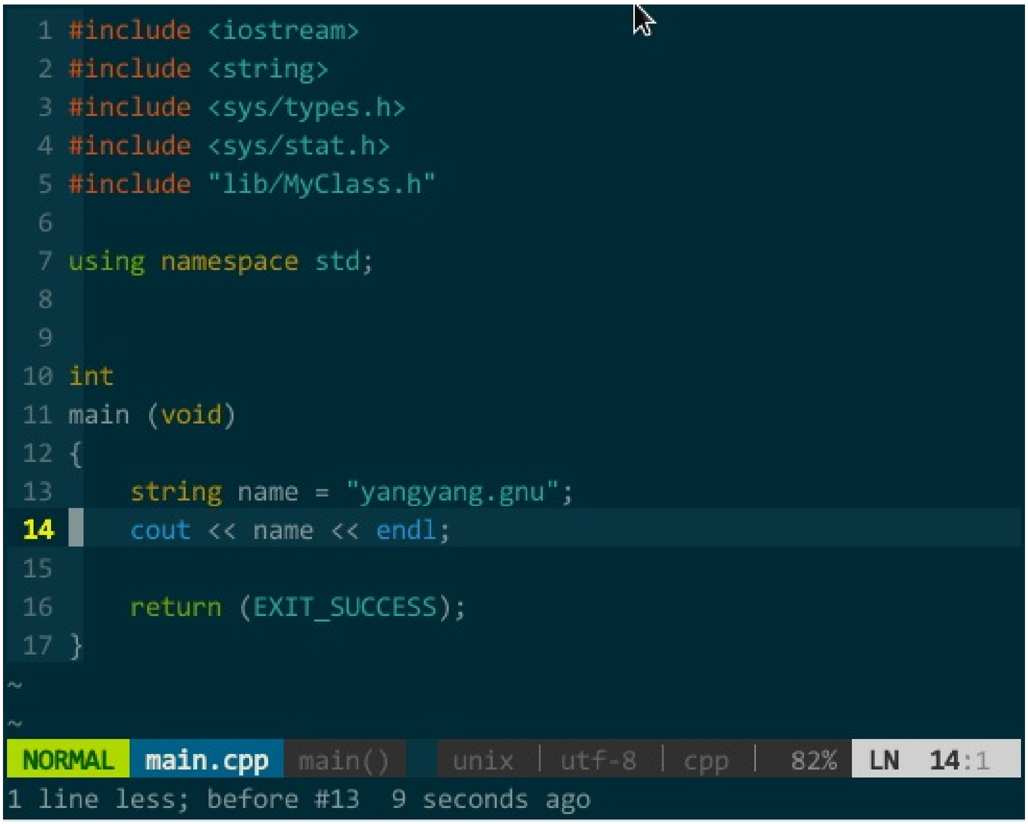


（YCM 的标签补全）

YCM 的 OmniCppComplete 补全引擎。我要进行 linux 系统开发，打开系统函数头文件觉得麻烦（也就无法使用 YCM 的语义补全），引入系统函数 tag 文件又影响 vim 速度（也就无法使用 YCM 的标签补全），这种情况又如何让 YCM 补全呢？WOW，别担心，YCM 还有 OmniCppComplete 补全引擎，只要你在当前代码文件中 #include 了该标识符所在头文件即可。通过 OmniCppComplete 补全无法使用 YCM 的随键而全的特性，你需要手工告知 YCM 需要补全，OmniCppComplete 的默认补全快捷键为 \\，不太方便，我重新设定为 \;，如前面配置所示：

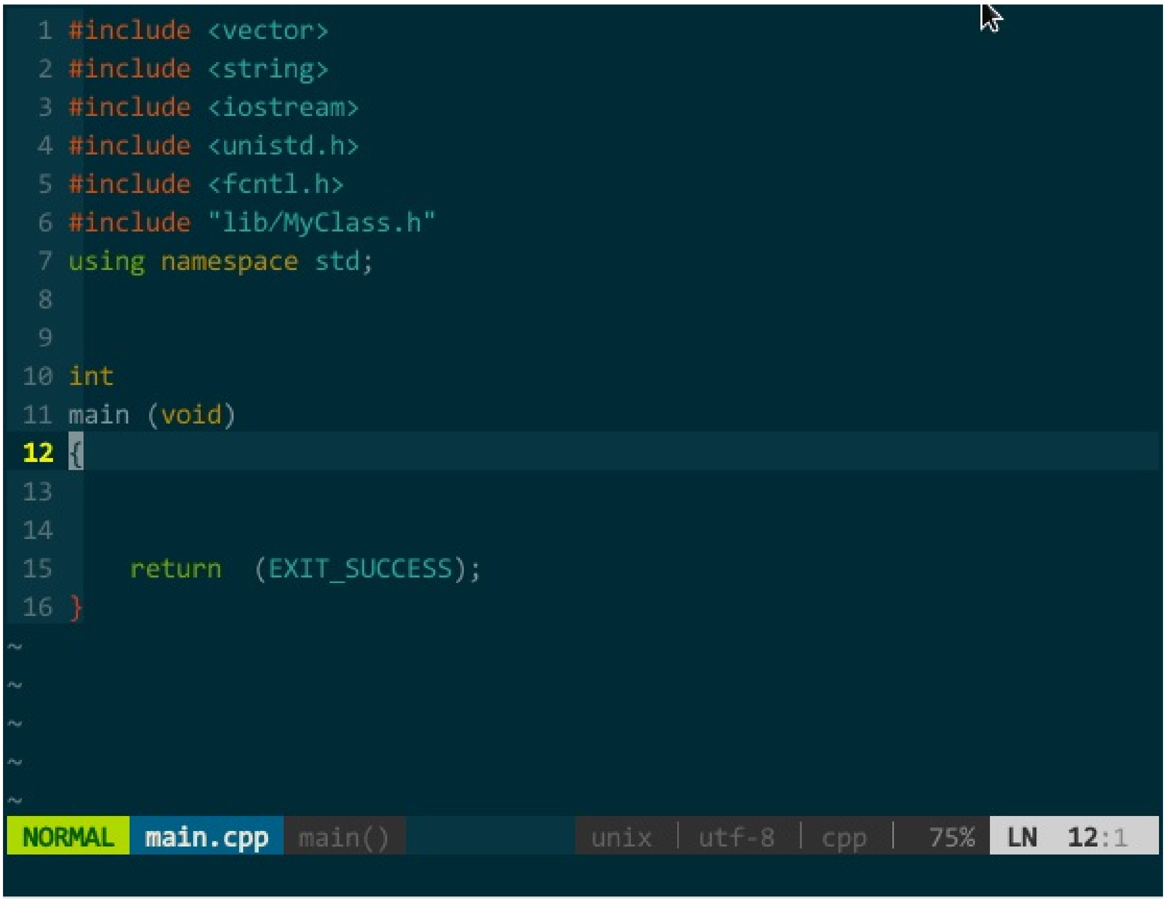
```
inoremap <leader>; <C-x><C-o>
```

比如，我要补全 fork()，该函数所在头文件为 unistd.h，正确添加 #include \ 后即可补全。如下图所示：



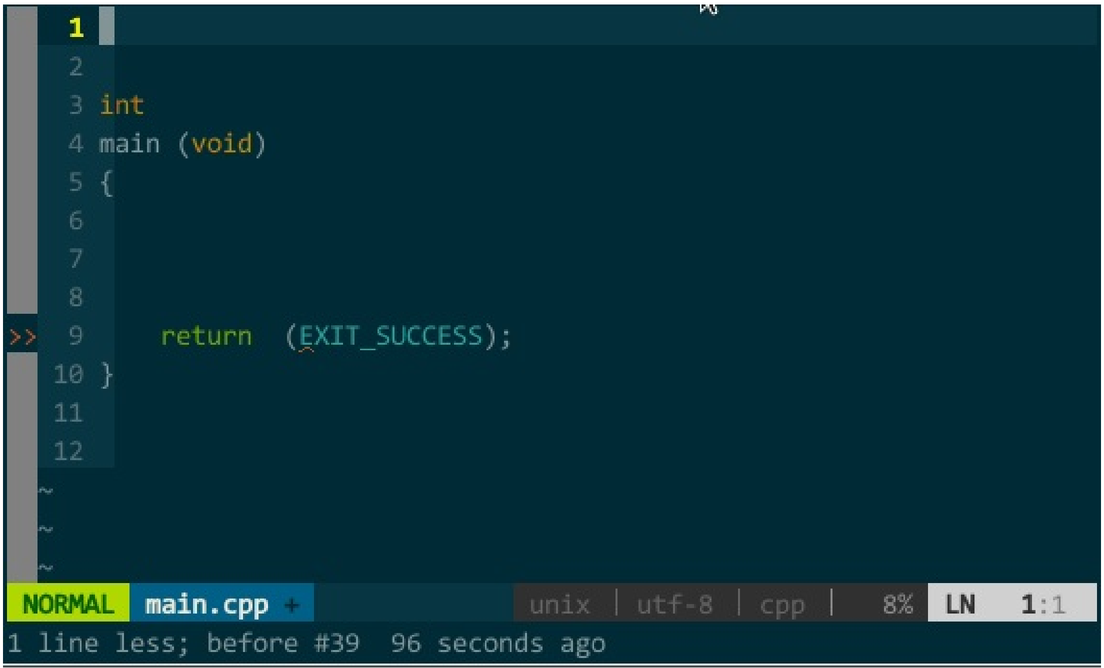
（YCM 的 OmniCppComplete 补全引擎）

其实，只要你正确插入头文件，YCM 的 OmniCppComplete 补全引擎可以替代语义引擎和标签引擎，比如，上面的 MyClass 在不打开 MyClass.h 情况下也可由OmniCppComplete（键入 \;）补全：



（OmniCppComplete 替代语义、标签补全）

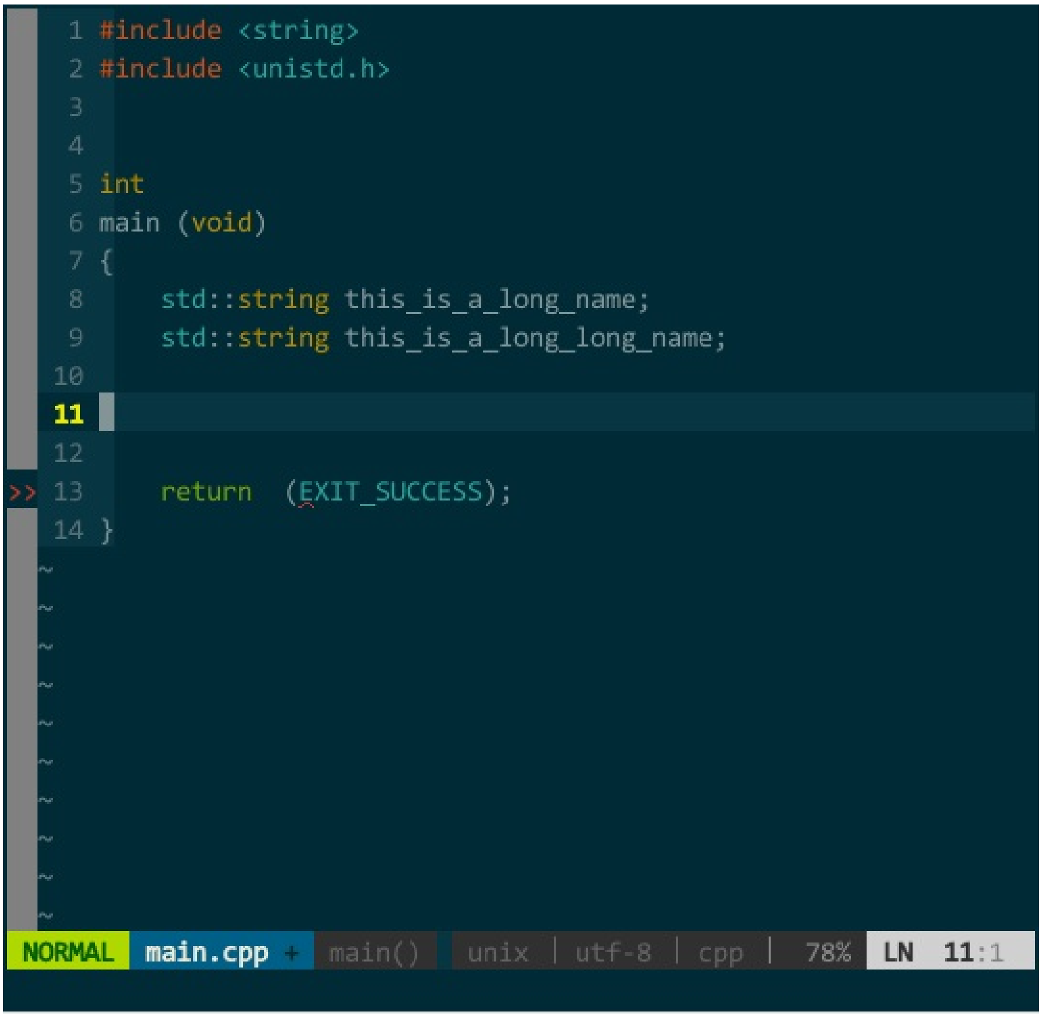
YCM 的其他补全。YCM 还集成了其他辅助补全引擎，可以补全路径、头文件、甚至可以在注释中继续补全。如下图：



（YCM 的其他补全）

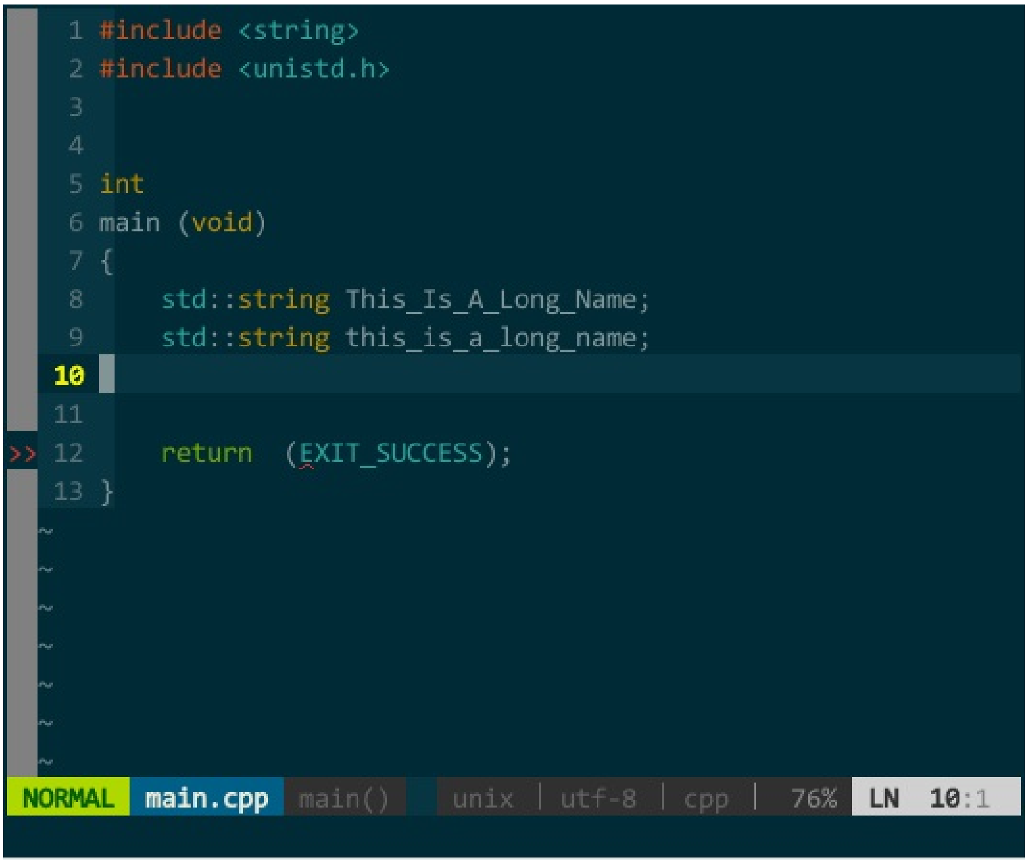
从我的经验来看，要想获得最好的补全体验，你应综合使用 YCM 的各种补全引擎！

另外，YCM 不愧是 google 工程师开发的，它的匹配项搜索方式非常智能，你无须从前往后逐一输入，YCM 会对你输入的内容进行模糊搜索，如下：



（YCM 模糊搜索）

此外，YCM 对大小写也非常智能，当你输入全小写时 YCM 对大小写不敏感，当然输入中有大写字母时 YCM 对大小写敏感：

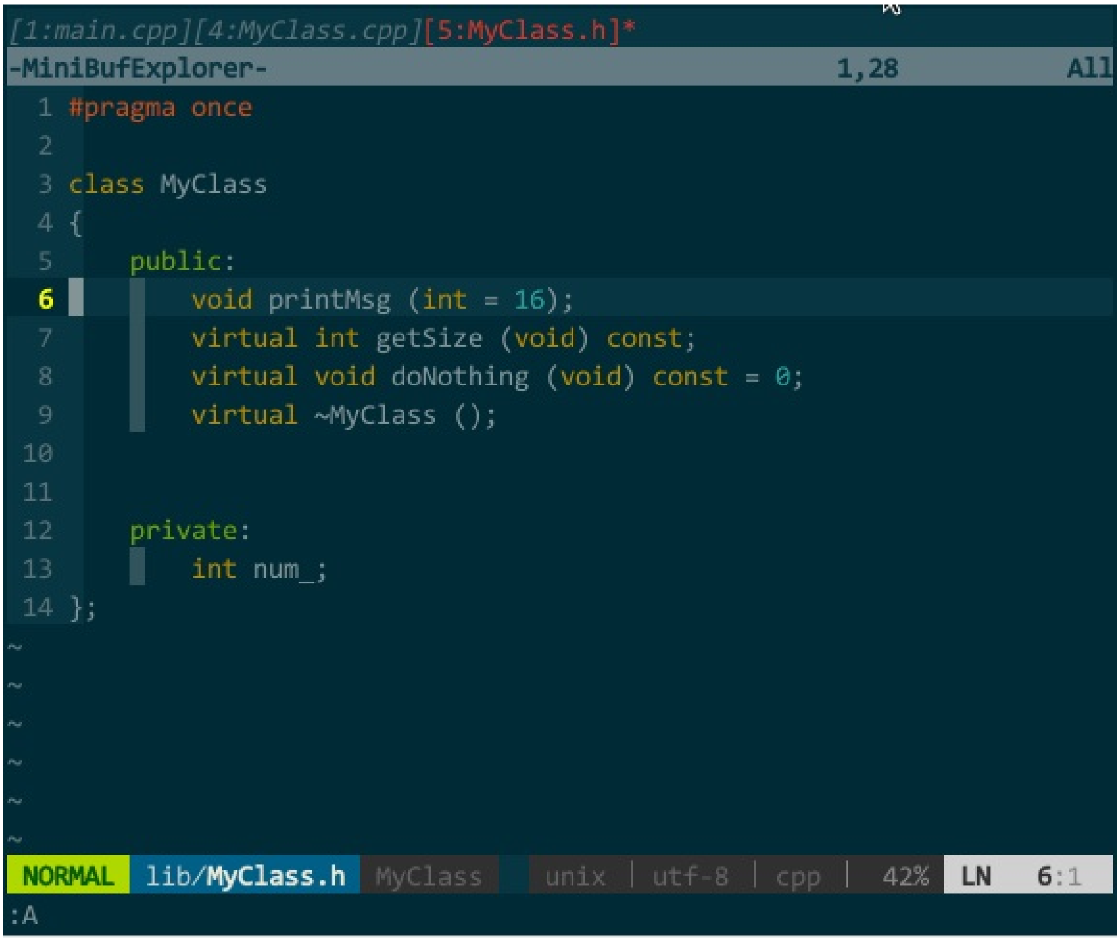


（YCM 大小写智能敏感）

上图中，当我键入 tia 时这两个对象均匹配，接着输入大写 L 时就只剩 This_Is_A_Long_Name 匹配。

5.4 由接口快速生成实现框架

在 *.h 中写成员函数的声明，在 *.cpp 中写成员函数的定义，很麻烦，我希望能根据类声明自动生成类实现的代码框架 —— vim-protodef（https://github.com/derekwyatt/vim-protodef ）。vim-protodef 依赖 FSwitch（https://github.com/derekwyatt/vim-fswitch ），请一并安装。请增加如下设置信息：``` " 成员函数的实现顺序与声明顺序一致 let g:disable_protodef_sorting=1 ``` protodef 根据文件名进行关联，比如，MyClass.h 与 MyClass.cpp 是一对接口和实现文件，MyClass.h 中接口为：``` " 设置 pullproto.pl 脚本路径 let g:protodefprotogetter='~/.vim/bundle/protodef/pullproto.pl' " 成员函数的实现顺序与声明顺序一致 let g:disable_protodef_sorting=1 ``` pullproto.pl 是 protodef 自带的 perl 脚本，默认位于 ~/.vim 目录，由于改用 pathogen 管理插件，所以路径需重新设置。 protodef 根据文件名进行关联，比如，MyClass.h 与 MyClass.cpp 是一对接口和实现文件，MyClass.h 中接口为：``` class MyClass { public: void printMsg (int = 16); virtual int getSize (void) const; virtual void doNothing (void) const = 0; virtual ~MyClass (); private: int num_; }; ``` 在 MyClass.cpp 中生成成员函数的实现框架，如下图所示：



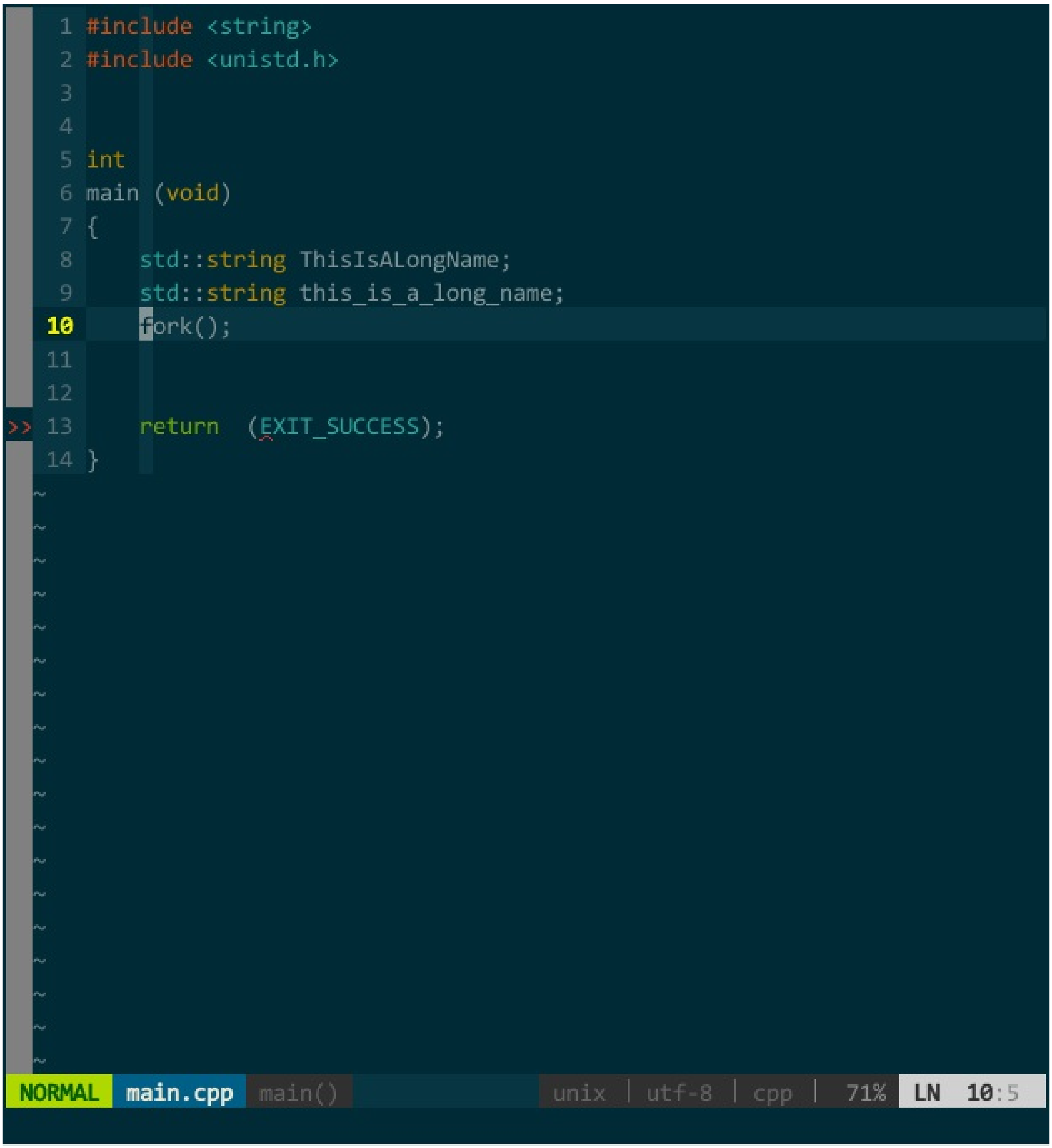
（接口生成实现）

MyClass.cpp 中我键入 protodef 定义的快捷键 \PP，自动生成了函数框架。 上图既突显了 protodef 的优点：优点一，virtual、默认参数等应在函数声明而不应在函数定义中出现的关键字，protodef 已为你过滤；优点二：doNothing() 这类纯虚函数不应有实现的自动被 protodef 忽略。同时也暴露了 protodef 问题：printMsg(int = 16) 的函数声明变更为 printMsg(unsigned)，protodef 无法自动为你更新，它把更改后的函数声明视为新函数添加在实现文件中，老声明对应的实现仍然保留。 关于缺点，先我计划优化下 protodef 源码再发给原作者，后来想想，protodef 借助 ctags 代码分析实现的，本来就存在某些缺陷，好吧，后续我找个时间写个与 protodef 相同功能但对 C++ 支持更完善的插件，内部当然借助 libclang 啦。 另外，每个人都有自己的代码风格，比如，return 语句我喜欢 ``` return(TODO); ``` 所以，调整了 protodef.vim 源码，把 239、241、244、246 四行改为 ``` call add(full, " return(TODO);") ``` 比如，函数名与形参列表间习惯添加个空格 ``` void MyClass::getSize (void); ``` 所以，把 217 行改为 ``` let proto = substitute(proto, '(_.*$', ' (' . params . Tail, '') ```

5.5 库信息参考

有过 win32 SDK 开发经验的朋友对 MSDN 或多或少有些迷恋吧，对于多达 7、8 个参数的 API，如果没有一套函数功能描述、参数讲解、返回值说明的文档，那么软件开发将是人间炼狱。别急，vim 也能做到。 要使用该功能，系统中必须先安装对应 man。安装 linux 系统函数 man，先下载（https://www.kernel.org/doc/man-pages/download.html ），解压后将 man1/ 至 man8/ 拷贝至 /usr/share/man/，运行 man fork 确认是否安装成功。安装 C++ 标准

库 man，先下载（ftp://GCC.gnu.org/pub/GCC/libstdc++/doxygen/ ），选择最新 libstdc++-api-X.X.X.man.tar.bz2，解压后将 man3/ 拷贝至 /usr/share/man/，运行 man std::vector 确认是否安装成功； vim 内置的 man.vim 插件可以查看已安装的 man，需在 .vimrc 中配置启动时自动加载该插件： ``` " 启用:Man命令查看各类man信息 source $VIMRUNTIME/ftplugin/man.vim " 定义:Man命令查看各类man信息的快捷键 nmap man :Man 3 ``` 需要查看时，在 vim 中键入输入 :Man fork 或者 :Man std::vector （注意大小写）即可在新建分割子窗口中查看到函数参考信息，为了方便，我设定了快捷键 \man，这样，光标所在单词将被传递给 :Man 命令，不用再手工键入，如下图所示：



（库信息参考）

另外，我们编码时通常都是先声明使用 std 名字空间，在使用某个标准库中的类时前不会添加 std:: 前缀，所以 vim 取到的当前光标所在单词中也不会含有 std:: 前缀，而，C++ 标准库所有 man 文件名均有 std:: 前缀，所以必须将所有文件的 std:: 前缀去掉才能让 :Man 找到正确的

man 文件。在 libstdc++-api-X.X.X.man/man3/ 执行批量重命名以取消所有 man文件的 std::前缀：

```
rename "std::" "" std::\*
```

顺便说下，很多人以为 rename 命令只是 mv 命令的简单封装，非也，在重命名方面，rename 太专业了，远非 mv 可触及滴，就拿上例来说，mv 必须结合 sed 才能达到这样的效果。

我认为，好的库信息参考手册不仅有对参数、返回值的描述，还应有使用范例，上面介绍的 linux 系统函数 man 做到了，C++ 标准库 man 还未达到我要求。所以，若有网络条件，我更愿意选择查看在线参考，C++ 推荐 http://www.cplusplus.com/reference/ 、http://en.cppreference.com/w/Cppreference:Archives ，前者范例多、后者更新勤；UNIX 推荐 http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html 、http://man7.org/linux/man-pages/dir_all_alphabetic.html ，前者基于最新 SUS（Single UNIX Specification，单一 UNIX 规范）、后者偏重 linux 扩展。

6 工程管理

我虽不要求达不到软件工程的高度，但基本的管理还是有必要的，比如，工程文件的管理、多文档编辑、工程环境的保存与恢复。

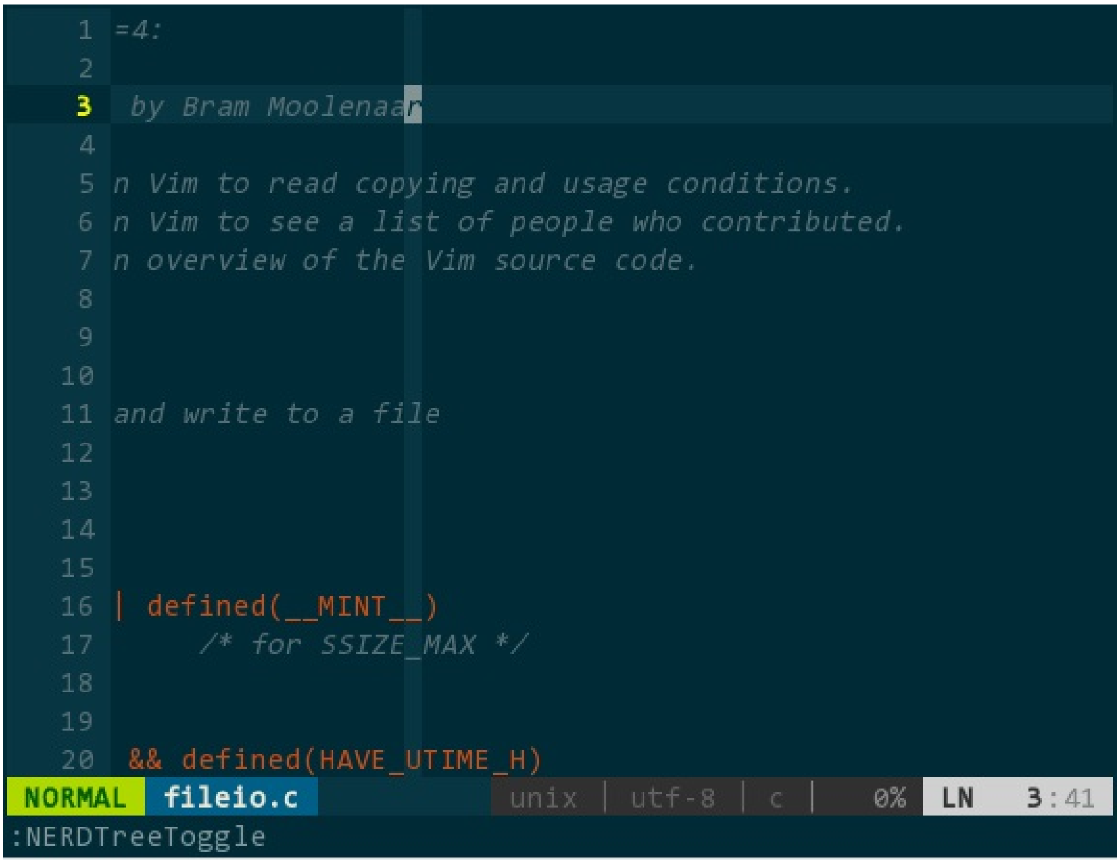
6.1 工程文件浏览

我通常将工程相关的文档放在同个目录下，通过 NERDtree（https://github.com/scrooloose/nerdtree ）插件可以查看文件列表，要打开哪个文件，光标选中后回车即可在新 buffer 中打开。

安装好 NERDtree 后，请将如下信息加入.vimrc中：

```
" 使用 NERDTree 插件查看工程文件。设置快捷键，速记：file list
nmap <Leader>fl :NERDTreeToggle<CR>
" 设置NERDTree子窗口宽度
let NERDTreeWinSize=32
" 设置NERDTree子窗口位置
let NERDTreeWinPos="right"
" 显示隐藏文件
let NERDTreeShowHidden=1
" NERDTree 子窗口中不显示冗余帮助信息
let NERDTreeMinimalUI=1
" 删除文件时自动删除文件对应 buffer
let NERDTreeAutoDeleteBuffer=1
```

常用操作：回车，打开选中文件；r，刷新工程目录文件列表；I（大写），显示/隐藏隐藏文件；m，出现创建/删除/剪切/拷贝操作列表。键入 \fl 后，右边子窗口为工程项目文件列表，如下图所示：

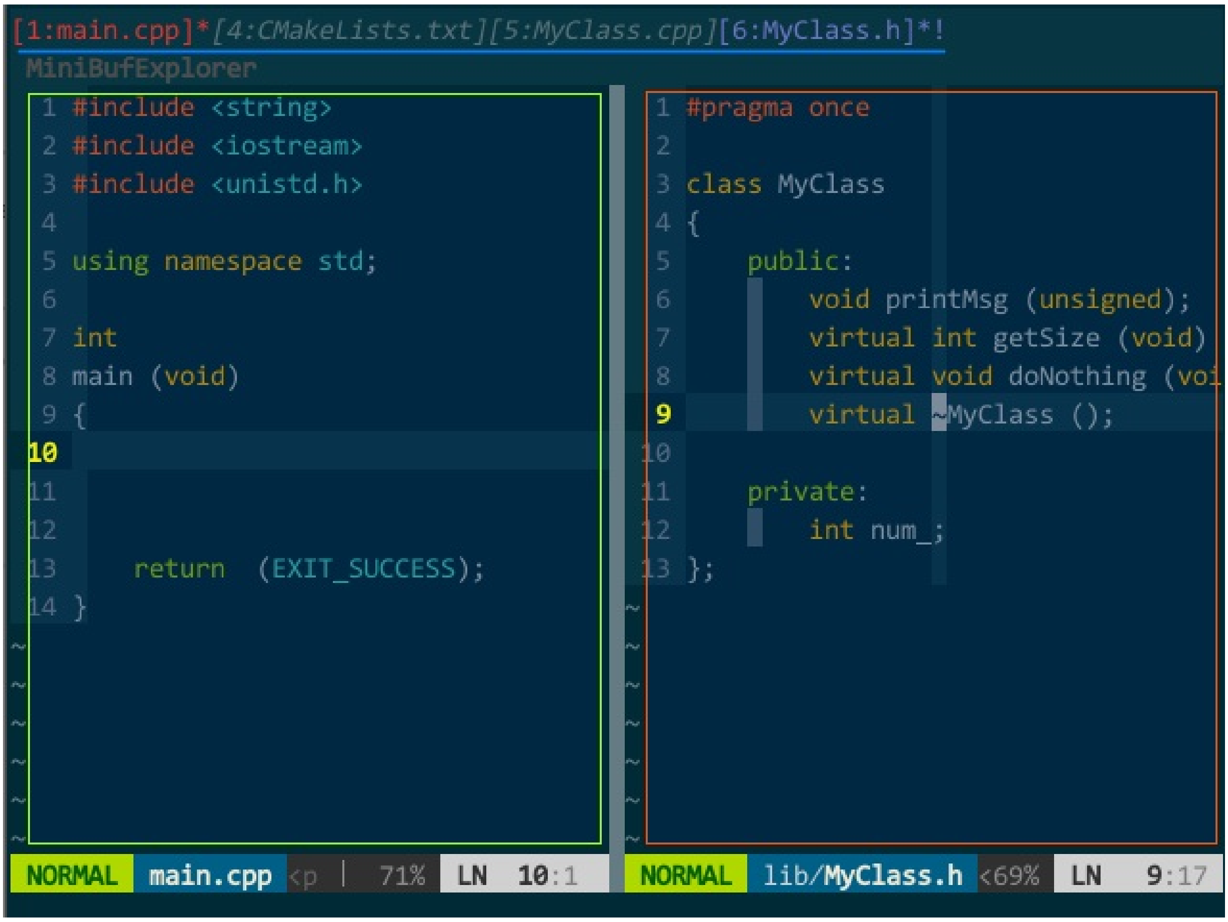


（工程文件浏览）

6.2 多文档编辑

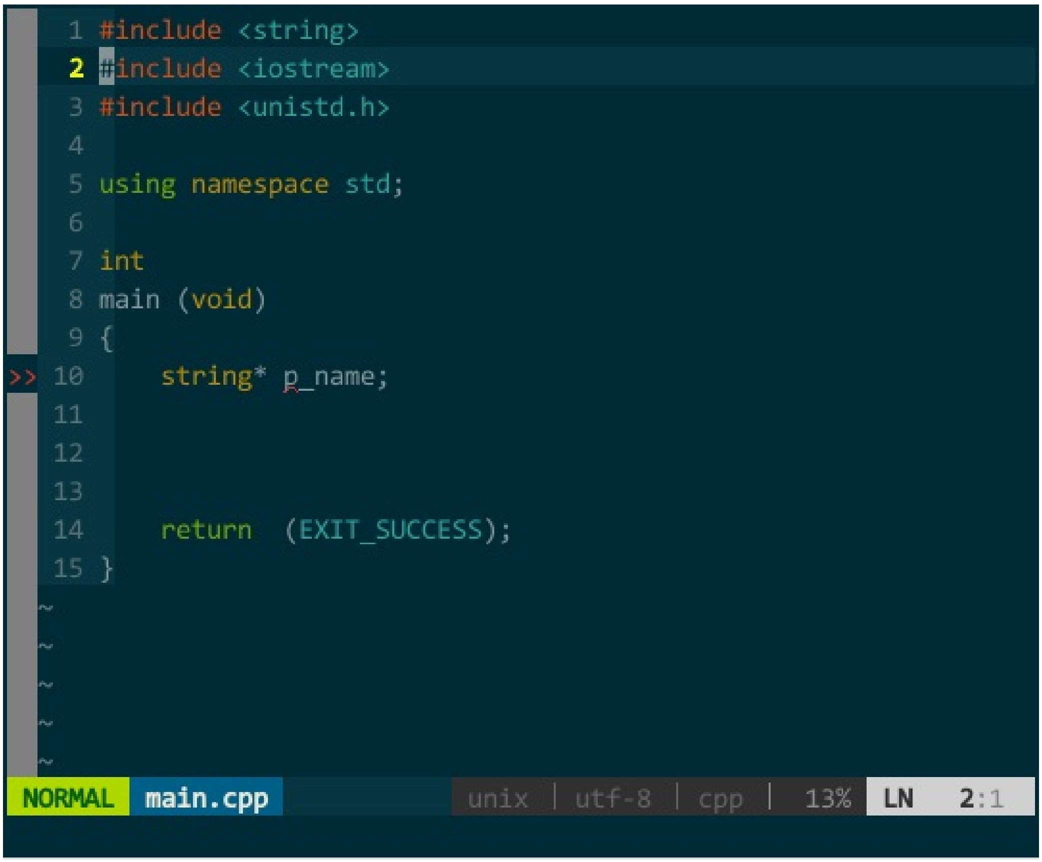
vim 的多文档编辑涉及三个概念：buffer、window、tab，这三个事物与我们常规理解意义大相径庭。vim 把加载进内存的文件叫做 buffer，buffer 不一定可见；若要 buffer 要可见，则必须通过 window 作为载体呈现；同个看面上的多个 window 组合成一个 tab。一句话，vim 的 buffer、window、tab 你可以对应理解成视角、布局、工作区。我所用到的多文档编辑场景几乎不会涉及 tab，重点关注 buffer、window。

vim 中每打开一个文件，vim 就对应创建一个 buffer，多个文件就有多个 buffer，但默认你只看得到最后 buffer 对应的 window，通过插件 MiniBufExplorer（https://github.com/fholgado/minibufexpl.vim ，原始版本已停止更新且问题较多，该版本是其他人 fork 的新项目）可以把所有 buffer 罗列出来，并且可以显示多个 buffer 对应的 window。如下图所示：



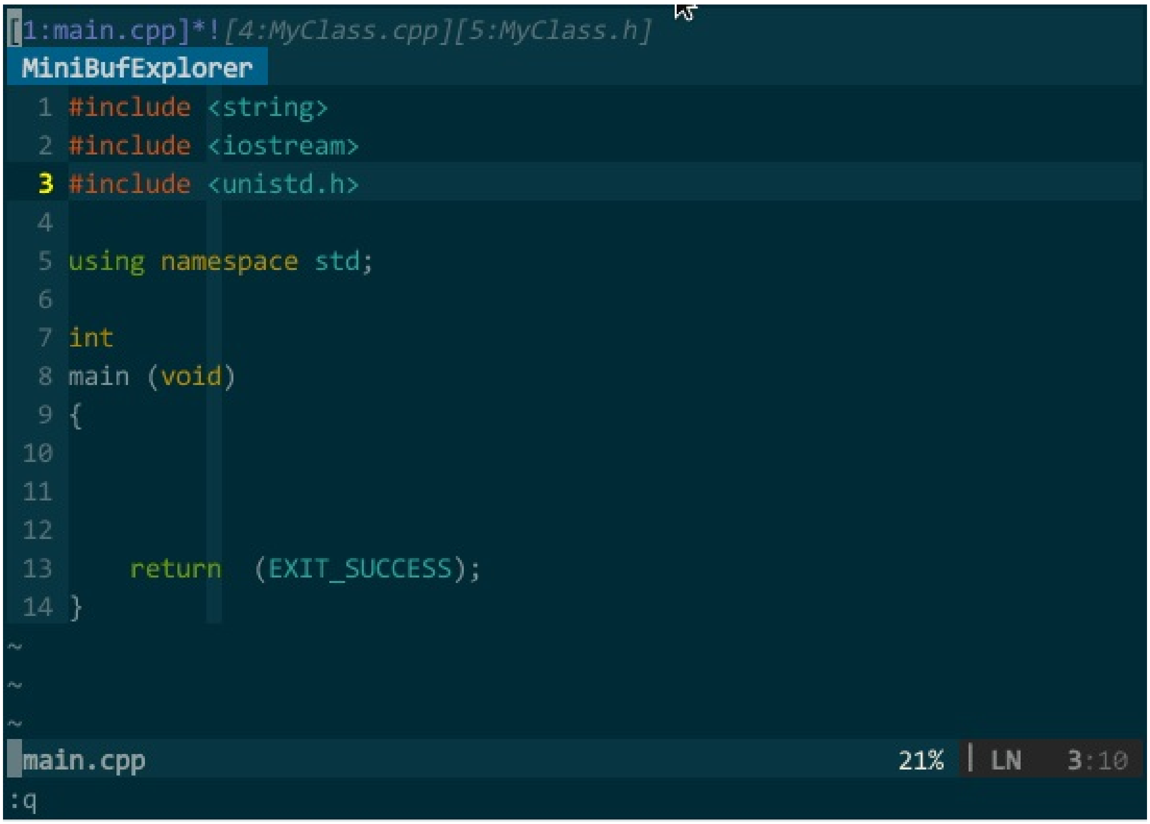
（buffer 列表）

我在 vim 中打开了 main.cpp、CMakeLists.txt、MyClass.cpp、MyClass.h 这四个文件，最上面子窗口（buffer 列表）罗列出的 [1:main.cpp][4:CMakeLists.txt][5:MyClass.cpp][6:MyClass.h] 就是对应的四个 buffer。当前显示了 main.cpp 和 MyClass.h 的两个 buffer，分别对应绿色区域和橙色区域的 window，这下对 buffer 和 window 有概念了吧。图中关于 buffer 列表再说明两点： * * 表示当前有 window 的 buffer，换言之，有 * 的 buffer 是可见的；! 表示当前正在编辑的 window； * 你注意到 buffer 序号 1 和 4 不连续的现象么？只要 vim 打开一个 buffer，序号自动增一，中间不连续有几个可能：可能一，打开了 1、2、3、4 后，用户删除了 2、3 两个 buffer，剩下 1、4；可能二，先打开了其他插件的窗口（如，tagbar）后再打开代码文件； 配置：将如下信息加入 .vimrc 中： ``` " 显示/隐藏 MiniBufExplorer 窗口 map bl :MBEToggle " buffer 切换快捷键 map :MBEbn map :MBEbp ``` 操作：一般通过 NERDtree 查看工程文件列表，选择打开多个代码文件后，MiniBufExplorer 在顶部自动创建 buffer 列表子窗口。通过前面配置，ctrl-tab 正向遍历 buffer，ctrl-shift-tab 逆向遍历（光标必须在 buffer 列表子窗口外）；在某个 buffer 上键入 d 删除光标所在的 buffer（光标必须在 buffer 列表子窗口内）：



（多文档编辑）

默认时，打开的 window 占据几乎整个 vim 编辑区域，如果你想把多个 window 平铺成多个子窗口可以使用 MiniBufExplorer 的 s 和 v 命令：在某个 buffer 上键入 s 将该 buffer 对应 window 与先前 window 上下排列，键入 v 则左右排列（光标必须在 buffer 列表子窗口内）。如下图所示：



（在子窗口中编辑多文档）

图中，通过 vim 自身的 f 名字查找 buffer 序号可快速选择需要的 buffer。另外，编辑单个文档时，不会出现 buffer 列表。

6.3 环境恢复*

vim 的编辑环境保存与恢复是我一直想要的功能，我希望每当重新打开 vim 时恢复：已打开文件、光标位置、undo/redo、书签、子窗口、窗口大小、窗口位置、命令历史、buffer 列表、代码折叠。vim 文档说 viminfo 特性可以恢复书签、session 特性可以恢复书签外的其他项，所以，请确保你的 vim 支持这两个特性：

```
vim --version | grep mksession
vim --version | grep viminfo
```

如果编译 vim 时添加了 --with-features=huge 选项那就没问题。

一般说来，保存/恢复环境步骤如下。

第一步，保存所有文档：

```
:wa
```

第二步，借助 viminfo 和 session 保存当前环境：

```
:mksession! my.vim
:wviminfo! my.viminfo
```

第三步，退出 vim：

```
:qa
```

第四步，恢复环境，进入 vim 后执行：

```
:source my.vim
:rviminfo my.viminfo
```

具体能保存哪些项，可由 sessionoptions 指定，另外，前面几步可以设定快捷键，在 .vimrc 中增加：

```
" 设置环境保存项
set sessionoptions="blank,buffers,globals,localoptions,tabpages,sesdir,folds,help,options
" 保存 undo 历史
set undodir=~/.undo_history/
set undofile
" 保存快捷键
map <leader>ss :mksession! my.vim<cr> :wviminfo! my.viminfo<cr>
" 恢复快捷键
map <leader>rs :source my.vim<cr> :rviminfo my.viminfo<cr>
```

这样，简化第二步、第四步操作。另外，sessionoptions 无法包含 undo 历史，你得先得手工创建存放 undo 历史的目录（如，.undo_history/）再通过开启 undofile 进行单独设置，一旦开启，每次写文件时自动保存 undo 历史，下次加载在文件时自动恢复所有 undo 历史，不再由 :mksession/:wviminfo 和 :source/:rviminfo 控制。

按此操作，并不能像 vim 文档中描述的那样能保存所有环境，比如，书签、代码折叠、命令历史都无法恢复。这和我预期存在较大差距，暂且用用吧，找个时间再深入研究！

7 工具链集成

既然我们要把 vim 打造成 IDE，那必须得集成编译器、构建工具、静态分析器、动态调试器，当然，你可能还需要版本控制、重构工具等等，我暂时还好。

7.1 编译器/构建工具集成

先说下编译器和构建工具。vim 再强大也只能是个优秀的编辑器而非编译器，它能高效地完成代码编辑工作，但必须通过其他外部命令实现将代码转换为二进制可执行文件；一旦工程上规模，你不可能单个单个文件编译，这时构建工具就派上场了。

代码编译

GCC 是 linux 上 C/C++ 编译器的事实标准，几乎所有发行套件都默认安装，它很好但不是最好：编译错误提示信息可读性不够（特别对于 C++ 模板错误信息基本就是读天书）、基于 GCC 的二次开发困难重重。我需要更优秀的 C++ 编译器。

Stanley B. Lippman 先生所推荐宇宙最强 C++ 编译器 —— LLVM/clang。Stanley 何许人也？不是吧，你玩 C++ 居然不认识他。C++ 世界二号人物，当年在贝尔实验室，Bjarne Stroustrup 构思了 C++ 功能框架，Stanley Lippman 实现了第一个版本。还无感？好吧，他是《C++ Primer》的作者。说了大神，再说说大神推荐的编译器。

LLVM 出自伊利诺伊大学研究项目，由 google 和苹果公司赞助。LLVM 的存在只为两个目的：一是尽可能地模块化现有代码以方便在此基础上进行二次开发、一是提供比传统构建工具链更好的用户体验。LLVM 是个很大很大的项目群，几乎把从编译到调试的各个构建环节都重新实现了一遍：

- 机器码生成方面：包含 LLVM core 子项目， LLVM core 能把满足它约定的中间语言翻译为高质量的机器码；
- 编译器方面：C/C++ 编译器 clang、接管 GCC 生成的 AST 并进行后续机器码生成的后端编译器 dragonegg；
- 调试器方面：增强处理模板/重载/多线程等等特性的调试器 LLDB、能根据程序 bug 生成测试用例甚至给出修正代码的符号虚拟机 klee；
- 链接器方面：能更好地处理符号链接的链接器 lld；
- 标准库方面：满足最新 C++ 规范的高性能实现标准库 libc++ 和 libc++ ABI、OpenCL 标准库的高性能实现 libclc；
- 运行期环境方面：支持 OpenMP 规范的运行期环境、Java 和 .NET 的虚拟机 vmkit；
- 代码优化方面：用于提升并行计算性能的 polly、针对内存安全检查的调优工具 SAFECode；

颤抖吧，小伙伴们！

我们重点介绍 clang 子项目，clang 把标准 C/C++ 代码转换为中间语言，换言之，前端 clang + 后端 LLVM（后简称 LLVM/clang）就是一款可替代 GCC 的优秀编译器。相较 GCC，LLVM/clang 有众多优势，尤其以下几点：

- 错误信息可读性强。能指出哪行、哪列有错误，并且用波浪线突显出来；另外，尽可能给出修改建议（比如提示你是否拼写错误）；最重要的是对 C++ 模板相关语法错误提示非常友好；（注，GCC 4.8 开始学习 clang 优化错误信息可读性）
- 编译速度快且占用资源少。编译速度是 GCC 的 2.5 倍，内存消耗只有 GCC 的 1/5；
- 兼容且扩充 GCC。clang 支持 GCC 的所有编译参数，也就是说，使用 GCC 开发的项目，你只需把 makefile 中使用的编译器从 GCC 改为 clang 即可，无须大面积调整构建系统脚本即可重新编译；另外，clang 还对 GCC 的编译参数进行了人性化扩展，比如，GCC 无法打开所有编译警告（-Wall、-Wextra 不够滴），clang 只需 -Weverything；
- 高度抽象的模块化设计。弱耦合性带来的模块高度复用、二次开发非常容易，比如，前面介绍的基于语义的 C/C++ 代码补全插件 YouCompleteMe，就是借助 libclang 库实现。

还在担心采用 clang 编译的源码移到 GCC 下无法编译？安啦，没问题的，你无非担心四方面：

- 编译参数是否兼容？前面说过，clang 全面兼容 GCC，所以编译参数完全兼容；
- 语言扩展是否兼容？只要不是像 linux 内核那样大规模采用各种复杂语言扩展属性，一般项目中用到的简单扩展是没问题的；
- 标准库接口是否兼容？标准库接口是 C++ 规范中指定的，根本不存在标准库接口不兼容一说；
- 标准库动/静态库符号是否兼容？你只要确保采用 clang 的标准库头文件对应链接 clang 的动态链接库、采用 GCC 的标准库头文件对应链接 GCC 的动态链接库的要求就不会出现问题。

在源码安装 clang 前，你需先自行安装 GCC，两个目的，一是你得有个编译器来编译编译器 clang （呵呵，绕了吧），二是其他人的项目可能会用到 GCC。

下载 LLVM、clang 及辅助库源码：

```
cd ~/downloads
# Checkout LLVM
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
# Checkout Clang
cd llvm/tools
svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
cd ../..
# Checkout extra Clang Tools
cd llvm/tools/clang/tools
svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra
cd ../../../..
# Checkout Compiler-RT
cd llvm/projects
svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
cd ../..
```

关掉其他应用，尽量多的系统资源留给 GCC 编译 clang 源码：

```
mkdir build
cd build
cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release ../llvm
make && make install
```

接下来，你先洗个澡，再约个会，回来应该编译好了（thinkpad T410I，CPU 奔腾双核 P6000，MEM 4G DDRIII，耗时 2 小时）。确认下：

```
clang --version
```

玩 C/C++ 你肯定要用到标准库。概念上，GCC 配套的标准库涉及 libstdc++ 和 libsupc++ 两个子库，前者是接口层（即，上层的封装），后者是实现层（即，底层的具体实现），对应实物文件，你得需要两个子库的头文件和动态链接库（*.so）。openSUSE 的安装源中有，直接安装头文件

```
zypper in libstdc++47-devel
```

位于 /usr/include/c++/4.7/，接着安装链接库

```
zypper in libstdc++6
```

位于 /usr/lib/libstdc++.so.6。呃，是滴，libstdc++、libsupc++ 两个子库的相关文件是合并一起安装的。

对应到 clang 的标准库，libc++（接口层）和 libc++abi（实现层）也需要安装头文件和动态链接库（*.so）。openSUSE 的安装源中并无 libc++，头文件和动态链接库只能源码安装：

```
cd ~/downloads/
svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx
cd libcxx/lib
./buildit
```

头文件已经生成到 ~/downloads/libcxx/include/，要让 clang 找到必须复制到 /usr/include/c++/v1/

```
cp -r ~/downloads/libcxx/include/ /usr/include/c++/v1/
```

*.so 文件已生成 ~/downloads/libcxx/lib/libc++.so.1.0，要让 clang 访问必须复制到 /usr/lib/，并创建软链接

```
ln -s ~/downloads/libcxx/lib/libc++.so.1.0 ~/downloads/libcxx/lib/libc++.so.1
ln -s ~/downloads/libcxx/lib/libc++.so.1.0 ~/downloads/libcxx/lib/libc++.so
cp ~/downloads/libcxx/lib/libc++.so* /usr/lib/
```

类似，源码安装 libc++abi 的头文件和动态链接库：

```
cd ~/downloads/
svn co http://llvm.org/svn/llvm-project/libcxxabi/trunk libcxxabi
cd libcxxabi/lib
./buildit
```

头文件已经生成到 ~/downloads/libcxxabi/include/，要让 clang 找到必须复制到 /usr/include/c++/v1/

```
cp -r ~/downloads/libcxxabi/include/ /usr/include/c++/v1/
\*.so 文件已生成 ~/downloads/libcxx/lib/libc++abi.so.1.0，要让 clang 访问必须复制到 /usr/lib/，
ln -s ~/downloads/libcxxabi/lib/libc++abi.so.1.0 ~/downloads/libcxxabi/lib/libc++abi.so.1
ln -s ~/downloads/libcxxabi/lib/libc++abi.so.1.0 ~/downloads/libcxxabi/lib/libc++abi.so
cp ~/downloads/libcxxabi/lib/libc++abi.so\* /usr/lib/
```

后续可以通过如下选项进行代码编译：

```
clang++ -std=c++11 -stdlib=libc++ -Werror -Weverything -Wno-disabled-macro-expansion -Wno
```

这一大波编译选项很崩溃么 @_@！我来简单说说：

- -std=c++11：使用 C++11 新特性；
- -stdlib=libc++：指定使用 clang 的标准库头文件 /usr/include/c++/v1/；
- -Werror：将所有编译警告视为编译错误；
- -Weverything：打开所有编译警告选项。在 GCC 中，无法通过单个选项打开所有编译警告，必须繁琐的同时指定 -Wall、-Wextra、以及大量分散的其他选项，为此 clang 新增了 -Weverything。

当然，有些警告意义不大，完全可忽略，如下：

- -Wno-disabled-macro-expansion：禁止使用宏表达式，忽略此警告；
- -Wno-float-equal：浮点类型不应使用 != 和 == 运算符，忽略此警告；
- -Wno-c++98-compat、-Wno-c++98-compat-pedantic：采用 C++11 新特性的代码无法兼容 C++98，忽略此警告；
- -Wno-global-constructors：在 main() 之前存在执行的代码，忽略此警告；
- -Wno-exit-time-destructors：在 main() 之后存在执行的代码，忽略此警告；
- -Wno-missing-prototypes：虽有函数定义但缺失函数原型，忽略此警告；
- -Wno-padded：结构体大小应为 4 字节整数倍，忽略此警告（编译器自动调整对齐边界）；

- -Wno-old-style-cast：C 语言的强制类型转换，忽略此警告；
- -lc++：指定链接 /usr/lib/libc++.so 标准库（缺失将导致链接失败！）；
- -lc++abi：指定链接 /usr/lib/libc++abi.so 标准库（缺失将导致链接失败！）。

系统构建

对于只有单个代码文件的项目来说，无非是保存代码文件、shell 中调用 GCC 编译、链接这样的简单方式即可实现；但，对于动辄几十上百个文件的工程项目，采用这种方式只会把自己逼疯，必须借助构建工具管理工程的整个构建过程。

linux 有两类工程构建工具 —— Makefile系 和非 Makefile 系，Makefile 系常见构建工具有 GNU 出品的老牌 autoconf、新生代的 CMake，非 Makefile 系中最著名的要数 SCons。KDE 就是通过 CMake（http://www.cmake.org/cmake/resources/software.html ）构建出来的，易用性灵活性兼备，洒泪推荐。

一般来说，你需要先写个名为 CMakeLists.txt 的构建脚本，然后执行 cmake CMakeLists.txt 命令将生成 Makefile 文件，最后执行 make 命令即可编译生成可执行程序。

举例来说，你工程包含 main.cpp 文件，要构建它，你需要执行如下步骤。

第一步，编写 CMakeLists.txt，内容如下：

```
PROJECT(main)
SET(SRC_LIST main.cpp)
SET(CMAKE_CXX_COMPILER "clang++")
SET(CMAKE_CXX_FLAGS "-std=c++11 -stdlib=libc++ -Werror -Weverything -Wno-deprecated-decla
SET(CMAKE_EXE_LINKER_FLAGS "-lc++ -lc++abi")
SET(CMAKE_BUILD_TYPE Debug)
ADD_EXECUTABLE(main ${SRC_LIST})
```

其中，PROJECT 指定工程名、SET 是 cmake 变量赋值命令、ADD_EXECUTABLE 指定生成可执行程序的名字。括号内的大写字符串是 cmake 内部预定义变量，这是 CMakeLists.txt 脚本的重点，下面详细讲述：

- SRC_LIST 指定参与编译的源码文件列表，如果有多个文件请用空格隔开，如，你工程有 main.cpp、lib/MyClass.cpp、lib/MyClass.h 三个文件，那么可以指定为：
- SET(SRC_LIST main.cpp lib/MyClass.cpp)
- CMAKE_CXX_COMPILER 指定选用何种编译器；
- CMAKE_CXX_FLAGS 设定编译选项；
- CMAKE_EXE_LINKER_FLAGS 设定链接选项。一定要将 -lc++ 和 -lc++abi 独立设定到 CMAKE_EXE_LINKER_FLAGS 变量中而不能放在 CMAKE_CXX_FLAGS，否则无法通过链接；
- CMAKE_BUILD_TYPE 设定生成的可执行程序中是否包含调试信息。

另外，对于编译选项，我的原则是严己宽人。也就是说，在我本机上使用最严格的编译选项以发现尽量多 bug，发布给其他人的源码包使用最宽松的编译选项以减少环境差异导致编译失败的可能。前面罗列出来的就是严格版的 CMakeLists.txt，宽松版我会考虑：编译器改用 GCC（很多人没装 clang）、忽略所有编译警告、让编译器进行代码优化、去掉调试信息、添加安装路径等要素，具体如下：

```
PROJECT(main)
SET(SRC_LIST main.cpp)
SET(CMAKE_CXX_COMPILER "g++")
SET(CMAKE_CXX_FLAGS "-std=c++11 -O3")
SET(CMAKE_BUILD_TYPE Release)
ADD_EXECUTABLE(porgram_name ${SRC_LIST})
INSTALL(PROGRAMS porgram_name DESTINATION /usr/bin/)
```

第二步，基于 CMakeLists.txt 生成 Makefile。在 CMakeLists.txt 所在目录执行：

```
cmake CMakeLists.txt
```

执行成功的话，你将在该目录下看到 Makefile 文件；

第三步，基于 Makefile 生成可执行程序。相同目录下执行：

```
make
```

这一步，就是在调用编译器进行编译，如果存在代码问题，修正错误后重新执行这一步即可，不用再次执行第一、二步。

基本上，你的新工程，可以在基于上面的 CMakeLists.txt 进行修改，执行一次第二步后，每次代码调整只需执行第三步即可。

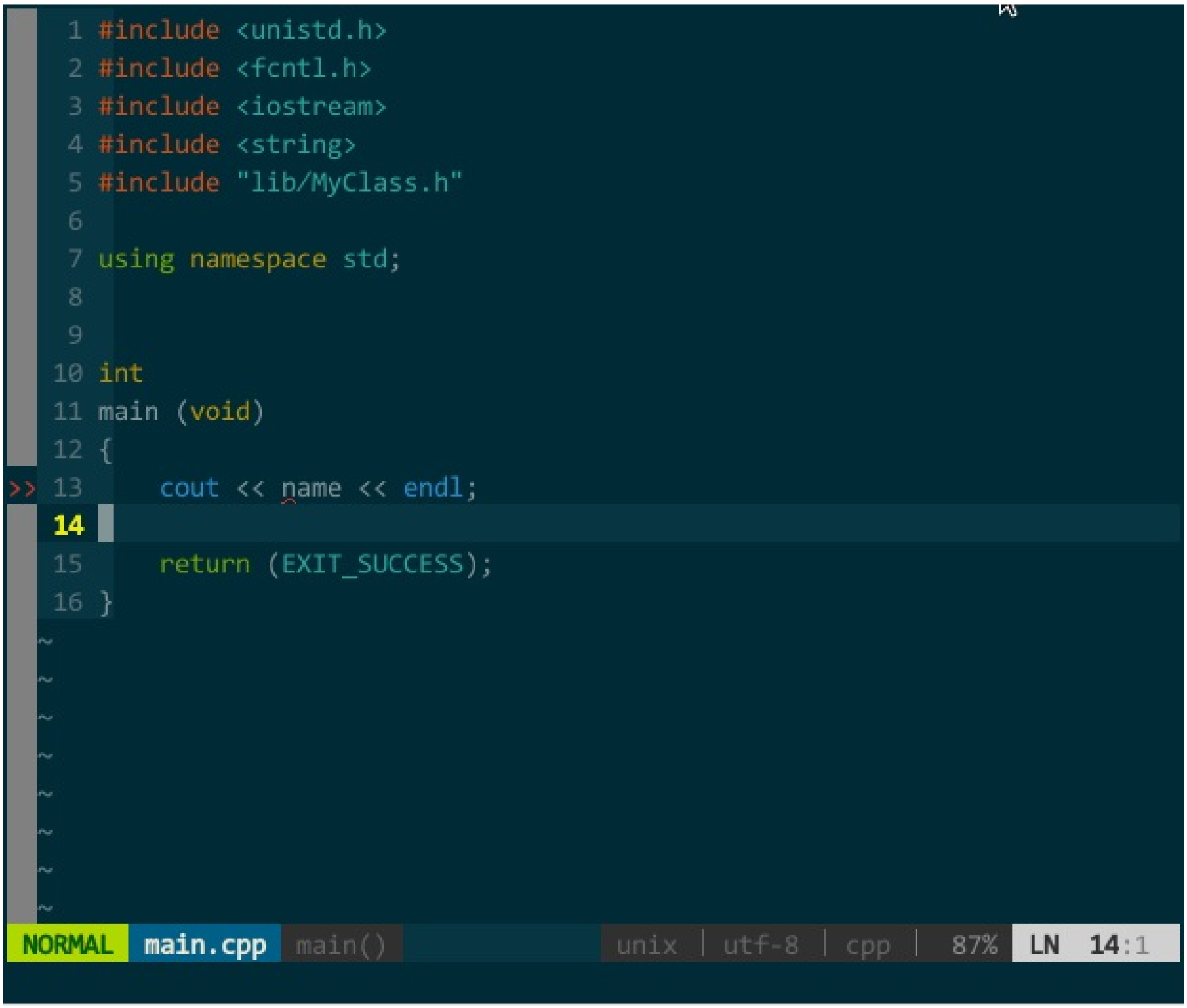
一键编译

工程项目的构建过程游离于 vim 之外终究不那么方便，前面章节介绍的构建过程是在 shell 中执行的，全在 vim 中执行又是如何操作。第一步的创建 CMakeLists.txt 没问题，vim 这么优秀的编辑器编辑个普通文本文件易如反掌；第二步的生成 Makefile 也没问题，在 vim 内部通过 ! 前缀可以执行 shell 命令，:!cmake CMakeLists.txt 即可；第三步的编译过程更没问题，因为 vim 自身支持 make 命令，直接在 vim 中输入 :make 命令它会调用外部 make 程序读取当前目录中的 Makefile 文件，完成编译、链接操作。当然，一次性编译通过的可能性很小，难免有些语法错误（语义错误只能靠调试器了），vim 将编译器抛出的错误和警告信息输出到 quickfix 中，执行 :cw 命令即可显示 quickfix。说了这么多，概要之，先通过构建工具（CMake 可通过 CMakeLists.txt 文件，autotools 可通过 configure 文件）生成整个工程的 Makefile，再在 vim 中执行 :make，最后显示 quickfix。

要实现一键编译，无非是把这几步映射为 vim 的快捷键，即：

```
nmap <Leader>m :wa<CR>:make<CR><CR>:cw<CR>
```

分解说明下，m 为设定的一键编译快捷键，:wa\ 保存所有调整文档内容，:make\ 调用 make 命令，后面的 \ 消除执行完 make 命令屏幕上“Press ENTER or type command to continue”的输入等待提示，:cw\ 显示 quickfix（仅当有编译错误或警告时）。如下图所示：



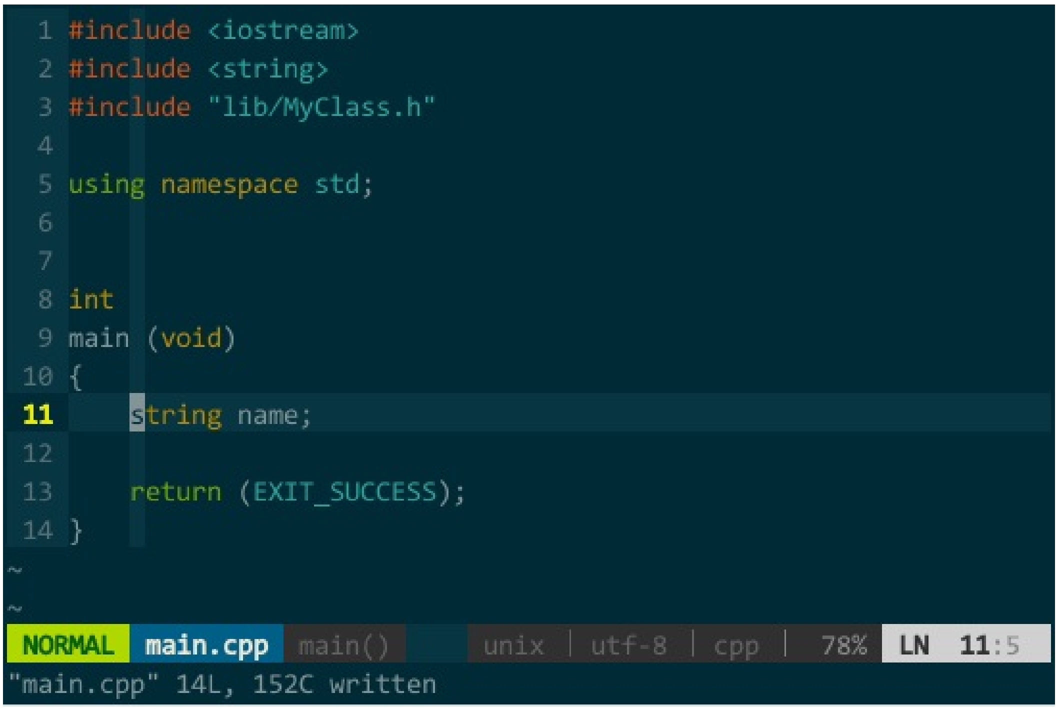
（一键编译）

我新建了一个工程，编辑好 CMakeLists.txt，执行 :!cmake CMakeLists.txt，接着 \m 一键编译，quickfix 窗口显示了编译错误，光标自动定位到需要你解决的第一个编译错误，回车后光标自动调整到该错误对应的代码位置，修正后重新 \r，编译通过并运行生成的程序。 你可能会遇到，调整过的代码能通过编译，但是，要么在工程目录中无法找到可执行程序，要么有程序但体现不出代码调整的内容（就像没调整过代码一样）。对于情况一，还算好，至少你晓得生成可程序失败了，肯定哪儿出了问题，不会继续往下新增代码；情况二，就麻烦了，你想通过运行程序检查刚才添加的代码运行是否正常，以为运行的是新程序，其实，代码调整后的新程序并未生成，运行是老程序，“哇，一切正常，往下写新业务逻辑代码”。导致这两个情况的根本原因，代码中存在链接错误导致并未正常创建新的可执行程序。bad news —— 如果编译错误，quickfix 窗口会固定在底部，罗列出所有编译过程中的所有错误，如果编译正常（即便是存在链接错误），quickfix 窗口会出现“Press ENTER or type command to continue”的输入等待提示信息，前面提过，为了省去手工输入回车，已经在 \m 中为 :make

多绑定个回车符 \，换言之，在编译正确链接错误的情况下，你是无法查看到 quickfix 窗口的；good news —— 有两种方式解决该问题：* 方式一，将前面 \m 中为 :make 绑定的回车符 \ 去掉，即 ``` nmap m :wa:make:cw ``` * 方式二，先删除老的可执行程序，再编译、链接，发现缺失可执行程序时，再手工执行 :make，这样，可查看具体是什么链接错误了，将如下配置信息加入 .vimrc 中： ``` nmap m :!rm -rf main:wa:make:cw ``` 我选方式二。 到此，已实现一键编译，要实现一键编译及运行无非就在刚才的快捷键中追加绑定运行程序的外部命令即可。新快捷键设定为 \g，假定生成的可执行程序名为 main，将如下配置信息加入 .vimrc 中： ``` nmap g :!rm -rf main:wa:make:cw:!./main ``` 最后，再次强调实现一键编译及运行的几个前提：vim 的当前目录必须为工程目录、事前准备好 Makefile 文件且放于工程目录的根目录、生成的程序必须在工程目录的根目录。

7.2 静态分析器集成

one take，中意“一次成型”，最早指歌手录歌时一次性通过录制，不存在发现错误-修正错误-重新录制这样的往返动作。one take 在编程环境中，就是一次性通过编译，我个人很享受 one take 带来的快感。当然，要达到 one take，不仅需要扎实的编程功底，还需要工具的辅佐 —— 代码静态分析器。前面介绍的神器 YCM 具备实时语法检查的能力。在它的作用下，编码中、编译前，所有语法错误都将被抓出来并呈现给你。 YCM 的整个静态分析过程分为如下几步： 第一步，发现错误。YCM 内部调用 libclang 分析语法错误，通过管道传递给 YCM 呈现。当你保存代码、vim 普通模式下移动光标或者安静 2 秒，错误检查后台任务将自动启动，若有错误，YCM 将接收到； 第二步，呈现错误。YCM 并不非立马显示错误信息，除非你触发下次击键事件，否则你看不到错误信息，换言之，干等是没结果的，你必须有次击键动作（没办法，vim 内部机制所限，后台任务无法直接更新 GUI，所以才采用变通的击键方式）。对于存在语法错误的代码，在行首有个红色的 >> 高亮显示； 第三步，查看错误。好了，现在已经知道哪行代码有问题，具体问题描述如何查看？两种方式：一种是将光标移至问题行，vim 将在其底部显示简要错误描述；一种是将光标移至问题行，键入 \d 后，vim 将在其底部显示详细错误描述。 如下所示：


1 #include <iostream>
2 #include <string>
3 #include "lib/MyClass.h"
4
5 using namespace std;
6
7
8 int
9 main (void)
10 {
11 string name;
12
13 return (EXIT_SUCCESS);
14 }
NORMAL main.cpp main() unix utf-8 cpp 78% LN 11:5
"main.cpp" 14L, 152C written

（静态代码分析）

8 其他辅助

大家关注的 IDE 核心功能前面都已逐一介绍过了，有些辅助功能我认为也有必要让你知道，不是都在提程序员人文关怀嘛，从我做起！

8.1 快速编辑结对符

平时，最让我头痛的字符莫过于 {}、""、[] 等这类结对符，输入它们之所以麻烦，主要因为 A）盲打很难找准它们位置，B）还得同时按住shift键。两者再一叠加，非常影响我的思维。要高效输入结对符，应该是输入少量几个字母（对，字母，不是字符）后 vim 自动为你输入完整结对符，而非是我输入一半 vim 输入另一半（不用 delimitMate 的原因）。刚好，这在 UltiSnips 能力范围内，只要定义好模板，可完美地解决这类问题，具体模板见上例中最后的结对符部分。

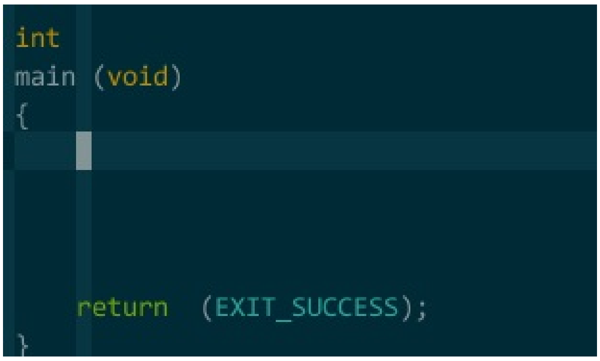
在定义结对符模板时，你应该考虑加上模板控制参数 i。默认情况下，UltiSnips 只会当模板名前是空白字符或行首时才进行模板补全，比如，定义 () 的模板如下：

```
snippet b "bracket"
(${1})${2}
endsnippet
```

我要调用函数 printf()，在输入完 printf 后应该接着输入括号模板名 b，然后输入模板展开快捷键 \\，你会发现 UltiSnips 无法帮你补全模板，因为它看到的不是 b 而是 printfb，这在模板文件中根本未定义。有一种间接解决方式是在 printf 后加个空格，再输入 b\\ 进行补全，这就成了 printf ()，不喜欢这种编码风格。其实，UltiSnips 的作者也注意到这个问题了，他让你可以通过前面提过的模板控制参数 i 进行解决。重新定义 () 的模板如下：

```
snippet b "bracket" i
(${1})${2}
endsnippet
```

这样，UltiSnips 只管光标前 1 个字符是否是 b，若是则补全 ()，不论 b 前是否有其他字符。类似，其他结对符模板都按此加上 i 控制参数。结对符模板完整定义参见上一节 cpp.snippets 示例。如下是几个快速输入结对符的演示：



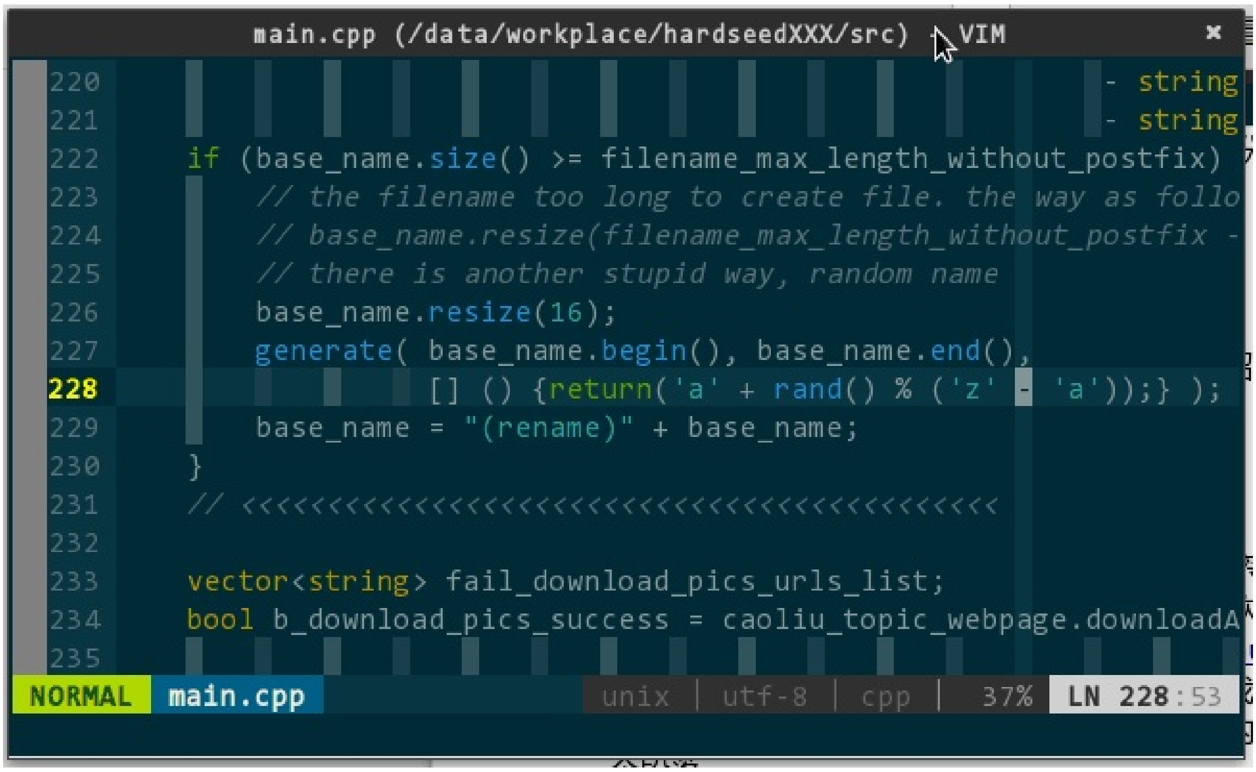
（快速输入结对符）

另外，要想高效编辑结对符，你得了解 vim 自身的某些快捷键。比如，有如下字符串且光标在该字符串的任意字符上，这时在命令模式下键入 va) 后将选中包括括号在内的整个字符串：



（快速选中结对符）

其中，v 是动作、a 是范围、) 是结对符。结对符命令的动作包括：选中 v、复制 y、删除 d、删除后插入 c；结对符命令的范围包括：含结对符 a、不含结对符 i。针对不同结对符，组合不同动作和范围就有 4*2 种方式。比如，di{ 删除不含结对符 {} 的字符串，va\[将选中含结对符 [] 内的所有字符。 选中结对符内的文本是我较为频繁的操作之一，通过诸如 vi\[的命令可以选中当前结对符 [] 内的所有文本，这虽谈不上麻烦，但每次都得去看下是]、)、> 还是 }，总是有点别扭。有款叫 wildfire.vim（https://github.com/gcmt/wildfire.vim ）的插件，让我能更自然地选中结对符内的文本，有了它，我只需按下空格（你也可以设置成其他快捷键），自动选中光标所在区域最近的一层结对符内的文本，如果没有结对符，则选择最近的一个段落。 简单设置： ``` " 快捷键 map (wildfire-fuel) vmap (wildfire-water) " 适用于哪些结对符 let g:wildfire_objects = ["i'", 'i"', "i)", "i]", "i}", "i>", "ip"] ``` 这样，在 vim 的命令模式下，一次空格选中最近一层结对符内的文本，两次则选中近两层内的文本，三次三层，依此类推；或者键入 3space，直接选中三层内的文本；若要取消，键入 shift-space 即可。另外，结对符类型也可以在 wildfire_objects 变量中指定。如下图所示：

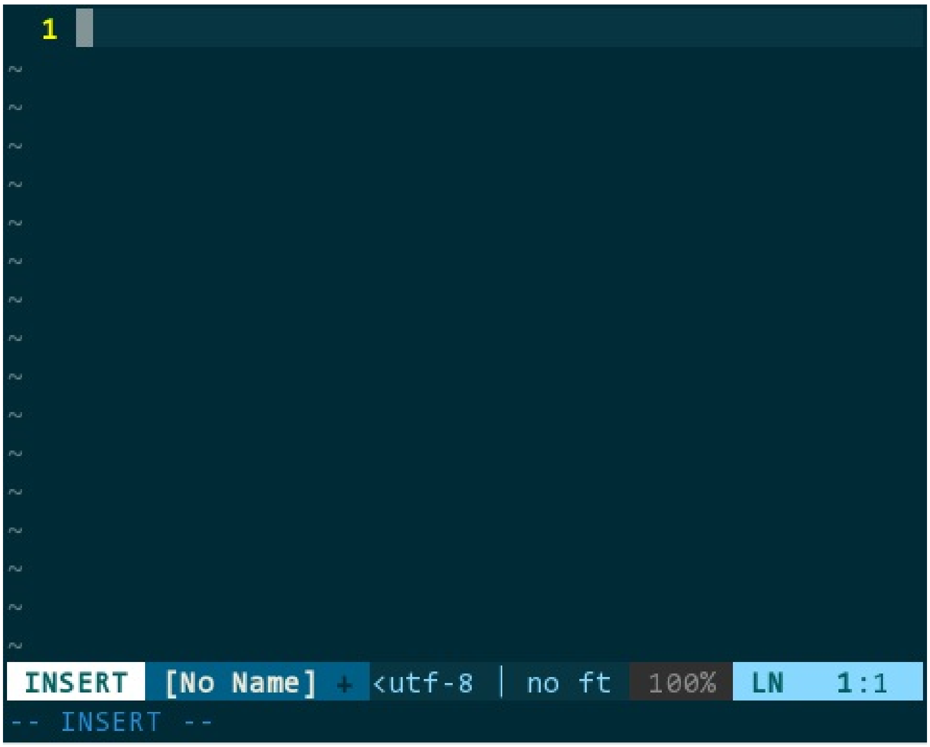


（快速选中结对符文本）

8.2 支持分支的 undo

undo，编辑器世界中的后悔药，让你有机会撤销最近一步或多步操作，这是任何编辑器都具备的基础功能。比如，第一步输入 A，第二步输入 B，第三步输入 C，当前文本为 ABC，一次 undo 后变成 AB，再次 undo 后变成 A，显然，每次 undo 撤销的均是最后的一步操作，通常采用栈这种数据结构来实现 undo 功能，由于栈具有后进先出的特点，所以，功能实现起来非常自然且便捷，但同时，也引入了致命伤，无法支持分支上的 undo 操作。

还是前面的例子，分三步依次输入完 ABC 后，一次 undo 变成 AB，这时，输入 D，之后，无论你多少次 undo 都不可能再找回 C，究其原因，D 是彻底覆盖了 C，而不是与 C 形成两个分支，如下图所示：



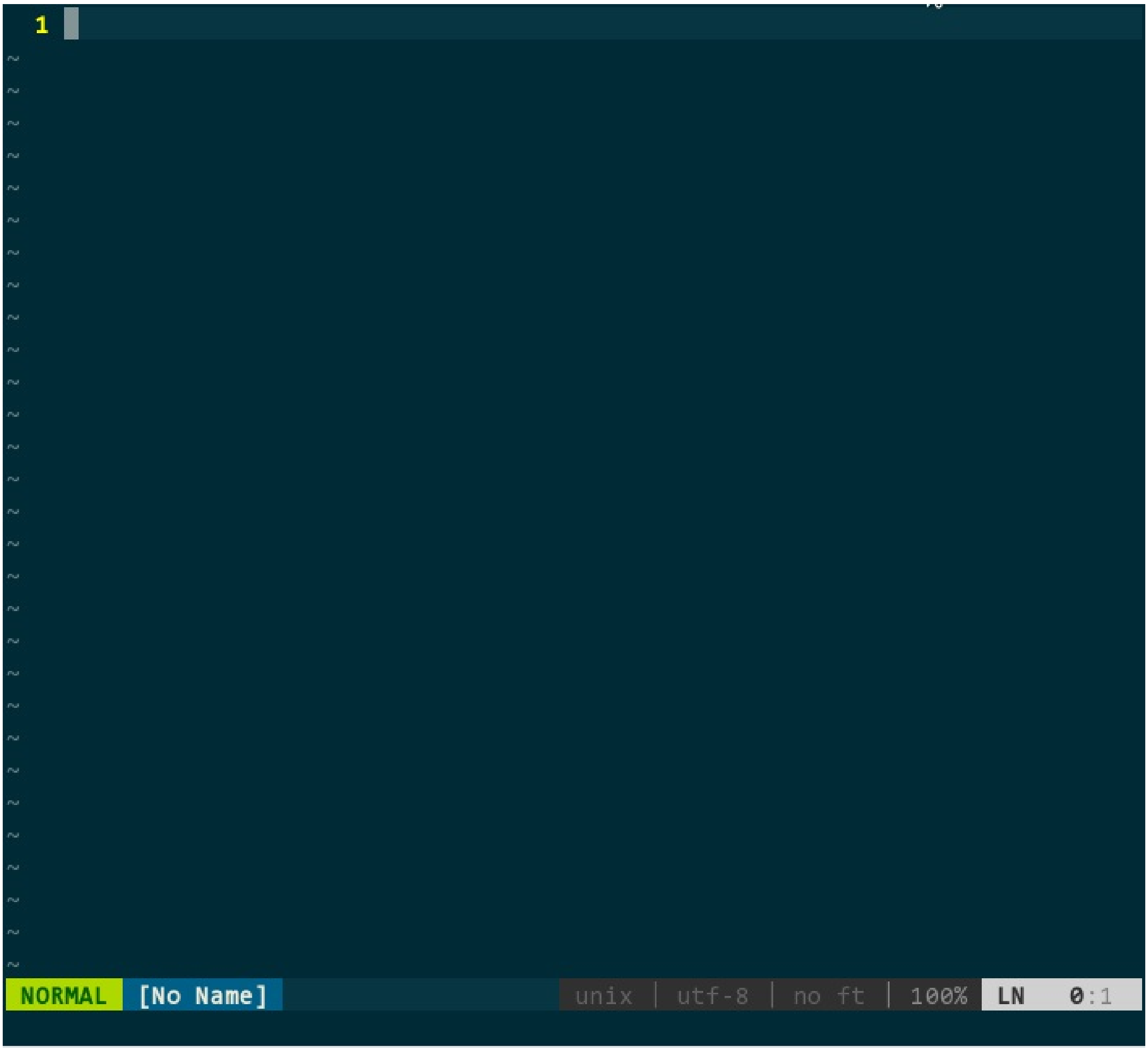
（不支持分支的 undo）

在我的使用场景中，非常需要在我输入 D 后还能找回 C 的 undo 功能，即，支持分支的 undo，gundo.vim （http://sjl.bitbucket.org/gundo.vim/ ）降临。gundo.vim 采用树这种数据结构来实现 undo，每一次编辑操作均放在树的叶子上，每次 undo 后，先回到主干，新建分支继续后续操作，而不是直接覆盖，从而实现支持分支的 undo 功能。gundo.vim 要求 vim 版本不低于 v7.3 且支持 python v2.4 及以上。

如下方式设置好调用 gundo.vim 的快捷方式：

```
" 调用 gundo 树
nnoremap <Leader>ud :GundoToggle<CR>
```

gundo.vim 非常贴心，调用它后，你会在左侧看到一个分割为上下两个区域的子窗口。上半区域以可视化方式显示了整颗 undo 树，同时，用 @ 标识最后一步编辑操作，用序号标识各步编辑操作的先后顺序，用时长显示每步操作距离当前消耗时间。下半区域展示了各个操作间的 diff 信息及其上下文，默认为选中那步操作与前一步操作间的 diff，键入 p 可以查看选中那步操作与最后一步操作（即有 @ 标识的那步）间的 diff，这对于找回多次编辑操作之前的环境非常有用。



（支持分支的 undo）

另外，我对持久保存 undo 历史也有需求，以便让我关闭 vim 后重新启动也能找到先前的所有 undo 历史，这需要你在 .vimrc 中添加：

```
" 开启保存 undo 历史功能
set undofile
" undo 历史保存路径
set undodir=~/.undo_history/
```

你得先自行创建 .undo_history/。具体可参见“6.3 环境恢复”章节。

8.3 快速移动

vim 有两类快速移动光标的方式：一类是以单词为单位的移动，比如，w 正向移动到相邻单词的首字符、b 逆向移动到相邻单词的首字符、e 正向移动到相邻单词的尾字符、 ge 逆向移动到相邻单词的尾字符；一类是配合查找字符的方式移动，比如，fa 正向移动到第一个字符 a

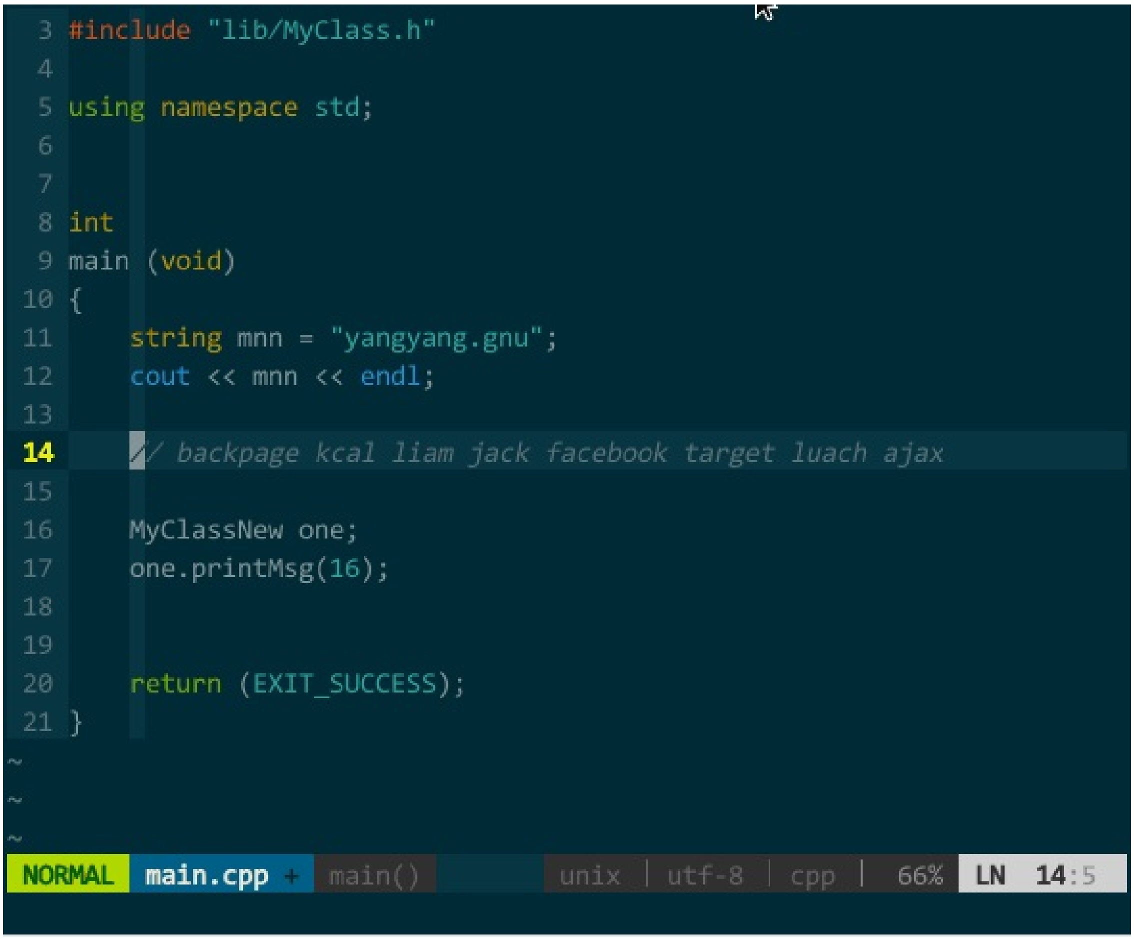
处、Fa 逆向移动到第一个字符 a 处。你要在非相邻的单词或字符间移动，你可以配合数字参数，比如，正向移动到相隔八个单词的首字符执行 8w、逆向移动到第四个 a 字符处执行 4Fa。

有如下文本：

> backpage kcal liam jack facebook target luach ajax

假定光标在行首，需要移动到 facebook 的字符 a 处，先来数下前面有 1、2 ... 5 个 a，然后用前面所说的 5fa，唔，怎么在 jack 上呢？等等，好像数错了，再数次 1、2 ... 6，对滴，应该是 6fa，这下对了。我的个天，不能让哥太累，得找个插件帮忙 —— easymotion（https://github.com/Lokaltog/vim-easymotion ）。

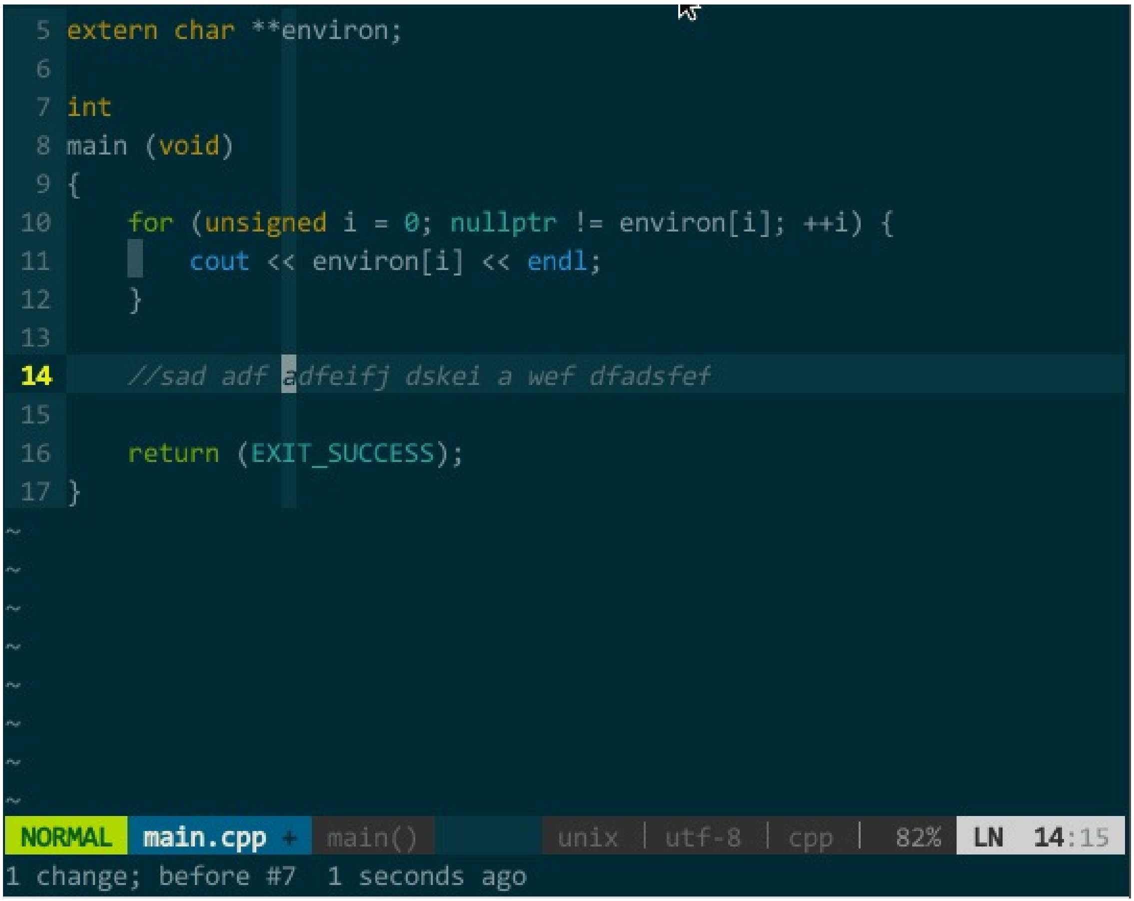
easymotion 只做一件事：把满足条件的位置用 [;A~Za~z] 间的标签字符标出来，找到你想去的位置再键入对应标签字符即可快速到达。比如，上面的例子，假设光标在行首，我只需键入 \\fa （为避免与其他快捷键冲突，easymotion 采用两次 \ 作为前缀键），所有的字符 a 都被重新标记成 a、b、c、d、e、f 等等标签（原始内容不会改变），f 标签为希望移动去的位置，随即键入 f 即可到达。如下图所示：



（快速移动）

类似，前面提过的 w、e、b、ge、F、j、k 等命令在 easymotion 作用下也能实现快速移动，其中，j 和 k 可跨行移动。同时，你还可以搭配 v 选中命令、d 删除命令、y 拷贝命令，比如，v\\fa，快速选中光标当前位置到指定字符 a 之间的文本，d\\fa，快速删除光标当前位置

到指定字符 a 之间的文本，下图所示：



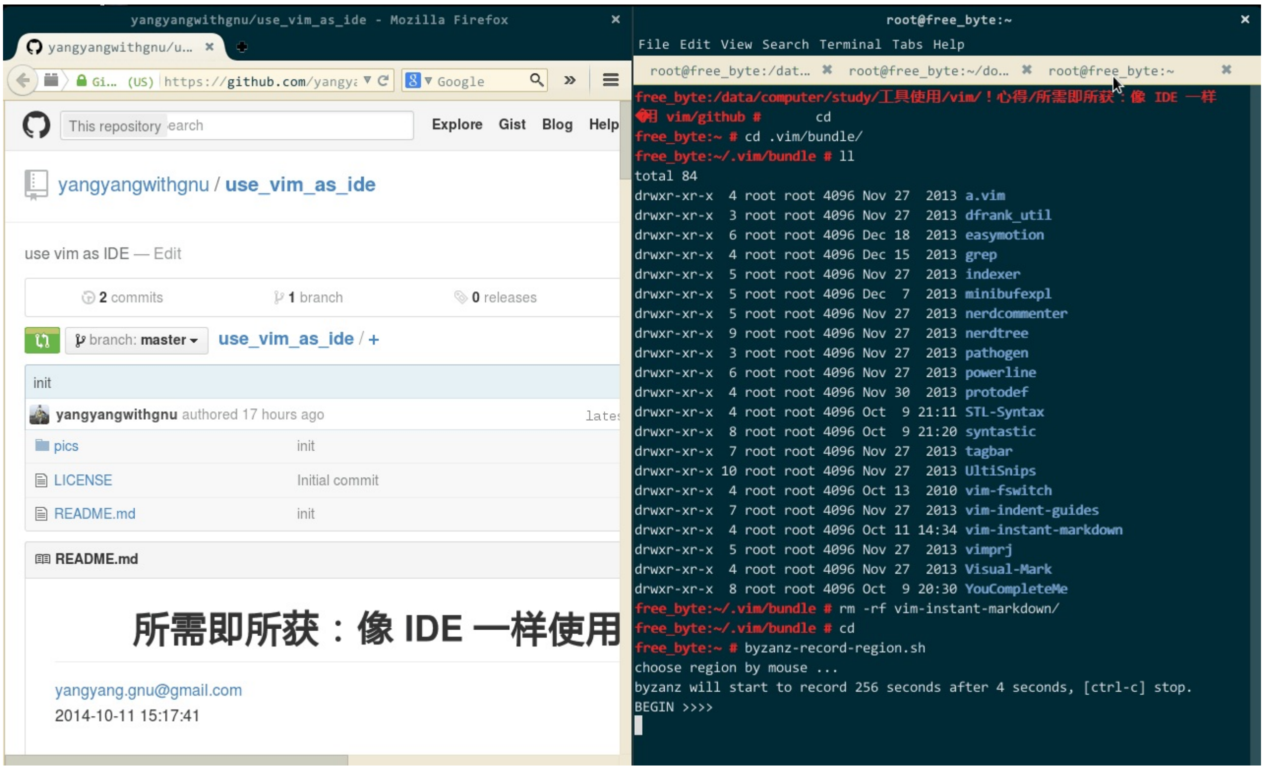
（搭配操作命令的快速移动）

8.4 markdown 即时预览

作为一枚热衷开源的伪 geek，github.com 与我同在。发布在 github.com 的任何项目你得有个项目简介，README.md，这是一种用 markdown 文书编写语法制作的说明文档。关于 markdown，我有两个需求，一是 vim 要高亮显示 markdown 语法，一是 firefox 要能渲染其语法、即时显示更新结果。

第一个需求不是问题，新版 vim 已经集成了 markdown 语法高亮插件，无须单独配置。

第二个需求，按照一般逻辑，应该通过 firefox 的某款插件来实现，的确，Markdown Viewer 看起来是干这个事儿的，但它响应速度缓慢、中文显示乱码、无法即时渲染等等问题让我无法接受。网上倒是有些即时渲染 markdown 的网站，比如，https://stackedit.io/editor ，左侧编辑右侧显示，所见即所得，但这又无法让我使用 vim，不行。还是回到 vim 身上想办法，vim-instant-markdown 来了。有了这款 vim 插件，一旦你启用 vim 编辑 markdown 文档，vim-instant-markdown 自动开启 firefox 为你显示 markdown 最终效果，如果你在 vim 中变更了文档内容，vim-instant-markdown 即时渲染、firefox 同步更新，太棒了！



（markdown 即时渲染）

vim-instant-markdown（https://github.com/suan/vim-instant-markdown ）的安装相比其他插件较为特殊，它由 ruby 开发，所以你的 vim 必须集成 ruby 解释器（见“1 源码安装编辑器 vim ”），并且安装 pygments.rb、redcarpet、instant-markdown-d 三个依赖库：

```
gem install pygments.rb
gem install redcarpet
# 若系统提示无 npm 命令，你需要先执行 zypper --no-refresh install nodejs
npm -g install instant-markdown-d
```

注意，以上三条命令均要翻墙。

对于重内容、轻设计的我这类人来说，markdown 简洁的文书语法太贴心了。推荐三个网站：

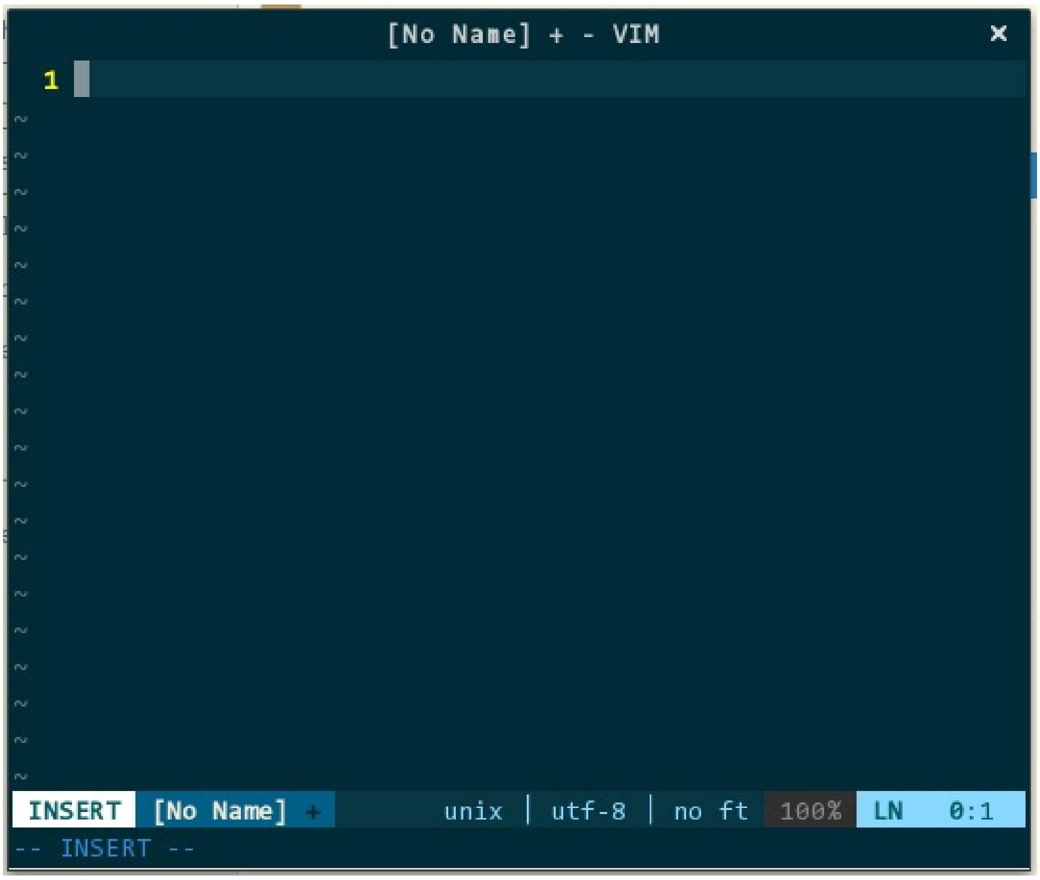
- markdown 语法 http://daringfireball.net/projects/markdown/syntax
- github.com 扩展 https://guides.github.com/features/mastering-markdown/
- emoji 符号表情 http://www.emoji-cheat-sheet.com/

8.5 中/英输入平滑切换

代码中不可能全是英文，即便注释是英文，用户提示信息也多多少少得用中文吧，所以，在 vim 中输入中文是常有的。中/英文输入切换本身很简单，但是，如果又与 vim 的插入模式和命令模式一纠缠，那么，这事儿就不太自然了。

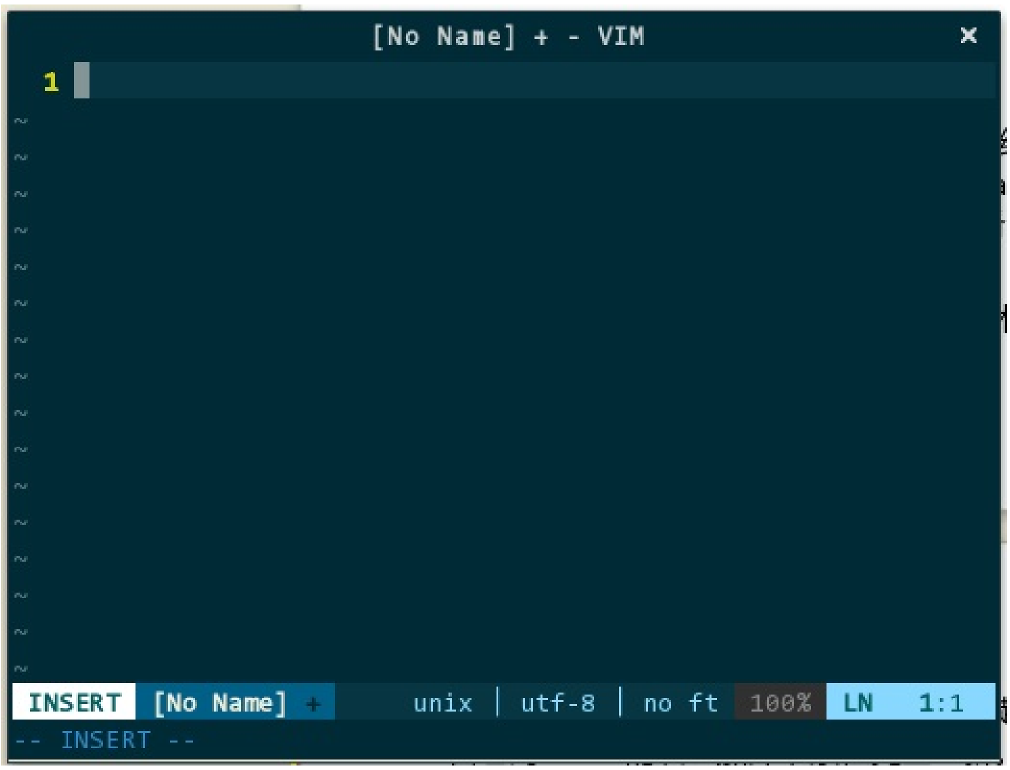
比如，我在插入模式下依次输入了中文的一二四三，本意是想输入一二三四，下意识地键入 esc 切换为命令模式，键入 x 剪切三，再键入 P 将三粘贴至四前。谁知，在键入 x 时，由于输入法仍保留在先前的中文状态下，导致 vim 的命令模式无法接收到命令，必须得再次键入

shift 切换至英文状态。如下图所示：



（中文状态让命令模式无效）

这很不协调，我希望，在插入模式下输入完中文，切换至命令模式后，即便先前是中文输入状态，也不影响我正常使用命令模式，甚至再次切回插入模式后，能保持先前的输入状态。来了，fcitx.vim（https://github.com/lilydjwg/fcitx.vim ）就是我要的。对，前提是你系统中用的是 fcitx 输入法（为何不用 scim、ibus？https://github.com/yangyangwithgnu/the_new_world_linux#7.4 ）。装好这个插件后，我们再看看刚才的例子，在中文状态下从插入模式切换至命令模式，键入 x、P 调整完四和三顺序后，重新切换至插入模式，输入法状态仍保持中文。如下图所示：



（即便中文状态也不影响命令模式）

几乎完美了，唯一问题是，该插件无法保证从其他程序窗口切换至 vim 后仍有效。

9 尾声

嗷呼，经过以上调教，你的 vim 已经成为非常舒适的 C/C++ 开发环境呢。等等，重装系统后又得折腾一次？不怕，除了 clang 等等几个需要源码安装的工具外，基本上，vim 的插件和相关配置文件你可以提前备份好，装完系统后恢复到对应目录中即可，丝毫不费脑力。

2011 年 9 月我写了篇《拼装的艺术：vim 之 IDE 进化实录》，原计划近期（2014-09）更新下智能补全部分，后来越改越发觉原版问题太多，加之各插件推陈出新、自己对 vim 的认识加深，索性完全重新。期间，与很多朋友有过交流，有三类问题探讨得最频繁，我的观点简要阐述如下，后续不再欢迎、理会、回复相关问题：

- 为何不用 Code::Blocks 这类一站式 IDE？每个人的做事的出发点、性格观念千差万别，我不想拿 linux kernel 是 linus torvalds 用 microemacs（emacs 变种）开发的来说事儿，就我而言，迷恋 vim 的高效编辑能力、无限扩充能力，这是其他编辑器无法超越的。此外，我享受的是过程，不是结果！
- 哪个是最适合编码的编辑器？linux 上存在两种编辑器：神之编辑器 emacs，编辑器之神 vim。关于 emacs 与 vim 孰轻谁重之争已是世纪话题，我无意参与其中，在我眼里，它们流淌着自由的血液、继承着创新的基因，作为顶级编辑器，二者在这个领域都作到了极致，让世人重新认识了编辑的本质 —— 用命令规则而非字字键入 —— 去完成编辑任务。emacs，伪装成编辑器的操作系统，太杂、太重，我更喜欢专注于编辑的 vim。
- 怎样的 IDE 才算好？对于初入开发的人员而言，Code::Blocks 是最易上手的选择；对于我这类喜欢折腾、追求效率、愿意用脑力换体力的人来说，vim 搭配各类插件是好的 IDE；对于 Donald Knuth 这等宗师，他们站在整个系统的层面，bash 加上几个命令行工具也是某种意义上的 IDE。所以，只要你能得心应手地完成软件开发任务，又察觉不到工具的存在，那这就是最适合你的 IDE。

末了，写作的过程，是知识体系完整重构的过程，理清了思路、加深了记忆。如果它再能引发你一点思绪，或许，这就是价值！